

Systematic Transaction Level Communication Modeling with SystemC

Von der Carl-Friedrich-Gauß-Fakultät
Technische Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades

Doktor-Ingenieur (Dr.-Ing.)

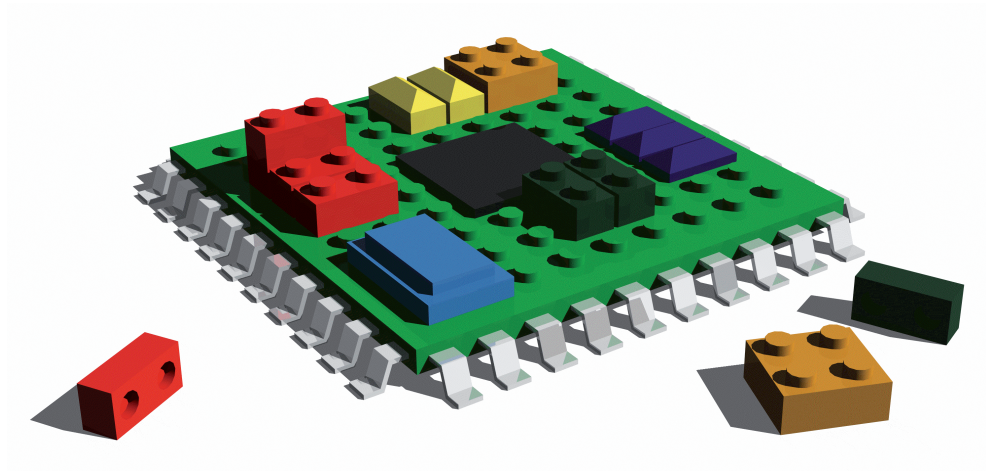
genehmigte Dissertation

von Dipl.-Inform. Wolfgang Klingauf
geboren am 15. April 1976 in Bonn

Eingereicht am:	13. Februar 2008
Mündliche Prüfung am:	7. Mai 2008
Referent:	Prof. Dr. Ulrich Golze
Korreferent:	Prof. Dr. Rolf Drechsler

(2008)

Systematic Transaction Level Communication Modeling with SystemC



Abstract

Embedded systems are becoming the most ubiquitous application of computer technology. Well-known examples are cell phones and digital cameras, while there are also more and more invisible embedded systems like distributed climate sensors in intelligent homes. To cope with the ever-increasing design complexity on a competitive basis, novel system level design tools are being offered, mostly based on the system description language SystemC, along with prefabbed hardware and software blocks (IP, intellectual property).

SystemC allows describing the communication among IPs in terms of abstract operations (transactions). The promise is that with transaction-level modeling (TLM), future systems-on-chip with one billion transistors and more can be composed out of IPs as simple than playing with LEGO bricks. However, reality is far out. In fact, each IP vendor promotes another proprietary interface standard and the provided design tools lack compatibility, such that heterogeneous IPs cannot be integrated efficiently. Thus, hoping for industry-wide interoperability seems futile.

In this PhD thesis, a novel generic interconnect fabric for transaction-level modeling with SystemC is presented tackling three major aspects. First, a review of the bus and IO structures that I have analyzed, which are common in today's systems-on-chip environments, and require to be modeled at a transaction level; second, my findings in terms of the data structures and interface APIs that are required in order to model those (and I believe other) buses and IO structures; and third, the surrounding infrastructure that I believe can, and should be in place to support the modeling of those buses and IO structures.

The proposed approach enables inter-operation between models of different levels of abstraction (mixed-mode), and models with different interfaces (heterogeneous components), with as little overhead as possible. An abstraction level formalism is developed to systematically support simulation of the state-of-the-art buses and networks-on-chip such as IBM CoreConnect and PCI Express at all levels of TLM abstraction. A layered architecture allows to simulate the communication architecture characteristics independent of the (proprietary) interfaces of the connected IPs, thereby enabling engineers to use different design tools conjointly for model analysis and debug. The thesis discusses new implementation techniques for communication modeling with SystemC, which outperform the existing approaches in terms of flexibility, simulation accuracy, and performance. Based on these techniques, advanced concepts for TLM-based high-level hardware/software co-design and FPGA prototyping are developed. Several experiments and a video processor case study highlight the efficiency of the approach and show its applicability in a TLM design flow.

The presented concepts have been submitted to the Open SystemC Initiative (OSCI) as interoperability standard proposal, and have been partially adopted in the new OSCI TLM 2.0 standard.

Kurzfassung

Eingebettete Systeme entwickeln sich zu dem wichtigsten Anwendungsgebiet in der Computertechnologie. Bekannte Beispiele sind alltägliche Geräte wie Handy und Digitalkamera, weniger bekannte sind unsichtbare Systeme wie verteilte Klimasensoren im intelligenten Haus. Um die immer kleineren und zunehmend vernetzten Produkte wettbewerbsfähig entwickeln zu können, haben sich System-Level-Entwurfswerkzeuge durchgesetzt. Entwickler können vorgefertigte Hardware- und Softwareblöcke im Quelltext einkaufen (IP, Intellectual Property) und diese mit Hilfe von Systembeschreibungssprachen zu einem Gesamtsystem (System-on-Chip) zusammensetzen.

Hier hat sich besonders die Sprache SystemC etabliert. Mit SystemC kann die Kommunikation zwischen IPs in Form abstrakter Operationen (Transaktionen) beschrieben werden. Die Hoffnung ist, dass mit der Transaction-Level-Modellierung (TLM) auch zukünftige Systeme mit 1 Milliarde Transistoren und mehr effizient entwickelt werden können. Idealerweise sollte das Hantieren mit hunderten IPs dabei so einfach sein wie das Spielen mit LEGO-Steinen. Die Realität ist allerdings weit davon entfernt. Jeder IP-Hersteller setzt derzeit auf einen anderen proprietären Schnittstellenstandard, so dass IPs unterschiedlicher Hersteller nicht ohne weiteres integriert werden können. Darüber hinaus sind die angebotenen Entwurfswerkzeuge nicht kompatibel, was zu einer Vielzahl von Insellösungen geführt hat, die firmenübergreifende Kooperationen erschweren.

In dieser Doktorarbeit wird ein neuer, generischer Ansatz für die Transaction-Level-Modellierung mit SystemC vorgestellt. Dabei stehen drei Hauptaspekte im Vordergrund. Erstens, eine Studie der Bus- und I/O-Kommunikationsstrukturen, die in heutigen System-on-Chip-Architekturen zum Einsatz kommen und mit Hilfe abstrakter Transaktionen modelliert werden sollen. Zweitens, meine Ergebnisse hinsichtlich der Datenstrukturen und Programmierschnittstellen, die für eine systematische Transaction-Level-Modellierung dieser Kommunikationsarchitekturen benötigt werden. Drittens, die Entwicklungsumgebung, die meiner Meinung nach zur Verfügung stehen sollte, um den IP-basierten Entwurf mit SystemC so effizient und intuitiv wie möglich zu unterstützen.

Der vorgestellte Ansatz ermöglicht Kommunikation zwischen Modellen auf unterschiedlichen Abstraktionsebenen (Mixed-Mode) und mit unterschiedlichen Schnittstellen (heterogene Komponenten). Der dazu benötigte Simulations- und Code-Overhead ist minimal. Es wird ein Abstraktionsebenen-Formalismus vorgestellt, mit dem verschiedenartige Busse und Networks-on-Chip wie IBM CoreConnect, PCI Express und CAN über alle TLM-Abstraktionsebenen simuliert werden können. Die Kommunikationsarchitektur-Simulation erfolgt dabei unabhängig von den (proprietären) Schnittstellen der angeschlossenen IPs und der verwendeten Entwurfswerkzeuge. Hierzu werden neue Implementierungstechniken diskutiert, die den existierenden Ansätzen in Flexibilität, Simulationsgenauigkeit, und -geschwindigkeit überlegen sind. Die Vor- und Nachteile der entwickelten Techniken werden mit Experimenten belegt, und eine Video-Prozessor-Fallstudie zeigt die Anwendbarkeit des Ansatzes in einem TLM-basierten Entwurfsfluss.

Die in dieser Arbeit vorgestellten Techniken wurden der Open SystemC Initiative (OSCI) als Vorschlag für einen SystemC-Interoperabilitätsstandard eingereicht und zum Teil in den neuen OSCI TLM 2.0 Standard übernommen.

Für Robin Benjamin

I would like to express my sincere gratitude to my advisor, Prof. Dr. Ulrich Golze. With his unique combination of sharp insight and encouragement, he made my years at EIS a both challenging and successful experience. His inspiring way to guide me to a deeper understanding of system level modeling was of invaluable help, and I always enjoyed the familiar yet thought-provoking atmosphere he creates. Furthermore, I would like to thank Prof. Dr. Rolf Drechsler from the University of Bremen for initiating a fruitful cooperation with his research group and for agreeing to be the co-examiner. I want to express my gratitude for his support and interest in my work. My thanks also go to Prof. Dr. Ursula Goltz for chairing the examination committee.

An extraordinary relationship has emerged through the help of Dan Burke, University of Illinois, and Adam Donlin from Xilinx, California, who exposed me to Mark Burton, now head of the GreenSocs initiative, Cambridge. Mark, please accept my heartfelt thanks for your extensive support and for promoting my research to the industry. I count myself lucky that I may consider you both a colleague and friend.

I have had the pleasure to work together on the Green* projects with some very talented students and colleagues. To the folks at EIS, FZI, Uni Bremen, Uni Campinas, and OFFIS: thanks for the great cooperation! Particularly, I would like to thank Robert Günzel and Christian Schröder, who started as my students and became the best teammates I can imagine. Special thanks go to Robert for always bringing me back down to earth when my imagination ran riot. Without all of them, this work would not be what it is now.

The most notable person, however, is Inga, my wife. Her strong interest in this work taught me how to explain technical stuff in an understandable and (sometimes) even women-inspiring manner, and her brilliant ability to condone my periodic ‘geekiness’ is second to none. Last but not least I would like to thank my parents Fred and Margit, who laid the groundwork for this and all other achievements in my live.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Stellen, die anderen Werken wörtlich oder sinngemäß entnommen sind, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat.

Wolfgang Klingauf

Contents

I	Efficient Communication Modeling with SystemC	1
1	Introduction	3
1.1	Scope and objectives of this work	4
1.2	Summary of contributions	6
1.3	Outline of the document	6
2	Basics of Transaction Level Modeling	7
2.1	Overview	7
2.1.1	Raising abstraction	7
2.1.2	Where is the level in TLM?	8
2.2	SystemC	10
2.2.1	Modules, ports, and channels	11
2.2.2	Events and the simulation semantics of SystemC	12
2.3	TLM terms and design flow	12
2.3.1	Structure of TLM models	12
2.3.2	The various roles of channels	14
2.3.3	On the notion of transactions	17
2.3.4	Conclusion	18
3	Towards a Generic TLM Fabric for SystemC and Related Work	19
3.1	Motivation	19
3.2	Communication modeling approaches for SystemC	21
3.2.1	User view	21
3.2.2	TLM view	22
3.2.3	Technical view	25
3.3	Survey of TLM frameworks	26
3.3.1	Summary	28
3.4	Discussion	28
3.4.1	Towards a TLM framework interoperability standard	30
4	The GreenBus Approach	33
4.1	Introduction	33
4.2	Requirements	34
4.3	General concepts	35
4.4	Bus accurate abstraction approach	37
4.4.1	Transactions, atoms, and quarks	37
4.4.2	Abstraction level formalism	41
4.5	A data representation and transport mechanism for TAQ	43
4.5.1	Data representation	43
4.5.2	Transport mechanism	44
4.5.3	Modeling communication delays	46
4.5.4	Copy at slave	48
4.6	Connection layer	48
4.6.1	Basic port	48
4.6.2	Payload event queue	49

4.6.3	Hierarchical ports	52
4.6.4	Concluding remarks	54
4.7	Generic protocol	56
4.7.1	Transaction container access	56
4.7.2	Generic communication	58
4.8	GreenBus extensions and interoperability	62
4.8.1	Extension mechanism	63
4.8.2	Compatibility	68
4.9	Adaptation layer	69
4.9.1	Implementation strategies for user APIs	69
4.10	Experiments	71
4.10.1	GreenBus optimization	72
4.10.2	Simulation performance	73
4.11	Summary and outlook	77
II Architecture Exploration and Analysis		79
5	Communication Architecture Exploration	81
5.1	A router architecture for GreenBus	81
5.1.1	Address map	81
5.1.2	PV bypass mode	83
5.1.3	BA/CC mode	83
5.2	Scheduling and arbitration	84
5.3	Protocol simulation	87
5.3.1	Mapping protocols onto atoms	89
5.4	Experiments	89
5.4.1	Processor Local Bus (PLB)	89
5.4.2	PCI Express	95
5.4.3	Network-on-Chip	95
5.5	Discussion of GreenBus CAFM accuracy	98
5.6	Summary	100
6	Performance Analysis and Visualization	101
6.1	Overview	101
6.2	GreenControl: a model instrumentation framework based on GreenBus	102
6.2.1	Requirements and related work	102
6.2.2	Transaction-based approach	103
6.2.3	Configurable parameters	103
6.2.4	Parameter database	103
6.2.5	User APIs	105
6.2.6	Tool support	105
6.2.7	Testbench creation using configuration files	106
6.3	DUST: a SystemC-aware design analysis toolkit	106
6.3.1	DUST backend	106
6.3.2	DUST frontend	107
6.3.3	Minimal-intrusive approach	107
6.3.4	XML based data processing	109
6.3.5	Compatibility	109
6.4	Summary and outlook	110
Conclusion		113
7	Summary and Future Work	113

Appendices	119
A Characteristics of SoC Communication Architectures	119
A.1 Overview	119
A.2 On-chip buses	119
A.3 Inter-chip buses	121
A.4 Bridges	122
A.5 Network-on-chip	122
A.6 Open Core Protocol	122
B User API Templates	123
B.1 Design patterns	123
B.1.1 MSBytesValid	126
B.1.2 Slave terminated transaction types	127
B.1.3 Error handling	127
B.2 State machines	127
B.2.1 Implementation	128
B.2.2 PV transactions	133
B.2.3 Abstraction switching	133
C EmViD Video Processor Case Study	135
C.1 Overview	135
C.2 High-level modeling and refinement with SHIP	136
C.2.1 GreenBus implementation of SHIP	138
C.2.2 Communication refinement for SHIP PEs	140
C.2.3 Back annotation of low-level timing	141
C.3 TRAIN and HPC	142
C.3.1 TRAIN architecture	142
C.3.2 Tools for software development	144
C.4 Architecture exploration and synthesis of a real-time gesture recognition system	144
C.4.1 Hardware/software partitioning and design flow parallelization	145
C.4.2 IP team	146
C.4.3 Hardware team	146
C.4.4 Software team	150
C.4.5 Integration and verification	151
C.4.6 System synthesis	151
C.5 Conclusion and outlook	154
Bibliography	157
Abbreviations	167
Index	169
Publications	177
Curriculum Vitae	179

Part I

Efficient Communication Modeling with SystemC

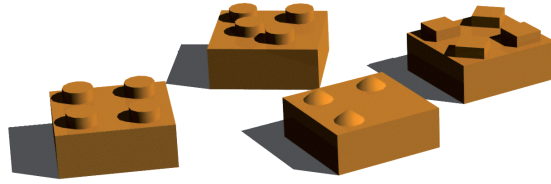


Figure 1.1: Play?

1 Introduction

A young start-up company from New York, Bug Labs, recently demonstrated what high-tech engineers can learn from LEGOTM: their product is an open-source modular hardware platform which lets customers build their own cell phone out of prefabbed building blocks. People can buy hardware ‘bricks’ such as GPS, digital camera, and touch-sensitive LCD screen, which, when plugged into one of the 3 sockets of the base module, are immediately accessible by the embedded software. When the company presented their approach in October 2007 [115], which is based on the open Java interface standard OSGiTM [100], the open-source community began to brew and first software applications became available for download just a few days later, written by volunteers around the world.

It would be the fulfillment of every computer engineers’ deepest dreams if the same ease became reality for embedded system design. Already today embedded systems regale us with a multitude of functionalities, integrated in an increasing range of articles of daily use. And future embedded systems will be even more embedded and chameleonic, becoming the ‘disappearing computer’ [130] of the century of ubiquitous computing.

With the increase in the number of logic gates that can be implemented on a single chip, a whole system previously integrated into a board is now being integrated into a single chip, with up to 100 million transistors and more. And cell phones, MP3 players, and digital cameras are only one outlier of this development. Embedded systems have now taken over the automobile. The luxury-car VW Phaeton boasts 61 different embedded control systems which are scattered over the car and do data exchange via four different in-vehicle networks.

System-on-chip (SoC) is the enabling technology of the 21st century [23]. It brings us a whole new range of products – small, intuitive, and communicative – and it drives an amazing revolution in how system engineers think and design. The keyword is *interoperability*. To cope with the complexity of networked embedded system design, engineers need to stop thinking in logic cells and wires. Instead, they must think in components [21, 111]. The goal is to enable embedded systems being so easily built from prefabbed hardware and software building blocks as playing with LEGO bricks.

However, the reality is shown in figure 1.1. There are so many interface ‘standards’ for SoC building blocks (typically referred to as intellectual property, IP) as companies who provide them. Most IPs are bound to a specific development tool, as it is the only tool that properly supports the IPs’ proprietary interfaces. Thus, SoC engineers are confined to the (limited) product range of a specific vendor, and hoping for interoperability seems futile.

In this PhD thesis a *generic interoperability standard* for systematic modeling of embedded systems with SystemCTM is developed that aims at overcoming this ‘interoperability gap’. GREENBUS, the communication fabric we have developed, implements this standard and enables composition of many IPs and joint use of different development tools, independent of their interfaces and abstraction levels.

A set of surrounding programming libraries and modeling techniques is presented that relieve the designer of the task of developing application-specific communication adapters and interface wrappers. My experiments and our results from industry cooperations with Intel, Texas Instruments, IBM, and Volkswagen show that this approach can enable faster exploration of design alternatives and contributes to design reuse and embedded software development.

The Bug Labs example illustrates the power of true interoperability. Bug Labs consumers are sure that their dearly bought components will seamlessly work together, and they can download and run any software made available by the open-source community. Thus, the ‘self-made’ phone becomes reality for everybody, as the ‘designer’ is able to concentrate on the functionality of his ‘product’, instead of brooding about low-level details. This makes the most important ingredient for successful products the main player in the design arena – creativity.

GreenSocs¹, an open-source initiative for system-level design tools, is devoted to exactly the same objective, but in the larger scope of system-on-chip design. With GREENBUS and the surrounding tools developed in this PhD thesis, which all have been made available as open source on the GreenSocs web page, this research aims at contributing to the ongoing challenge of identifying the ingredients for true LEGO-style embedded system design.

We have suggested a framework with which to build standardized adapters to permit designs with inhomogeneous IPs. Naturally, standards have to be adapted in the course of time, or will even produce anti standards. But we believe there today is a strong need for standardized and therefore unambiguous modeling techniques at the electronic system level (ESL). Engineers that step into this novel design approach need to be able to resort to well-approved methodologies in order to achieve straightforward results. However, such a standard must give ample scope for user extensions and creativity.

1.1 Scope and objectives of this work

If we want to believe the flurry of publications, raising the level of modeling abstraction to the electronics system level seems to be the silver bullet embedded system engineers were always looking for. The hope is that ESL design tools will soon allow the development of SoCs with up to one billion transistors and more [17, 92], based on a ‘drag-and-drop’ development style.

The complexity of networked SoCs introduces the problem of *communication modeling*. Their development requires the capability of modeling and simulating both their hardware and software components’ behavior/functionality (the *computation*) and the complex communication environment in which they operate (the *communication*).

Here, we focus on the communication. A bunch of different communications need to be adequately designed in order to ensure proper operation of the overall system:

- hardware-hardware communication between the processors, memories, network controllers, and application-specific hardware cores of a SoC;
- software-hardware communication for utilization of hardware services by the embedded software processes;
- software-software communication between the concurrent software processes of a SoC (who may reside on multiple processors);
- chip-chip communication between the different chips in an embedded system;
- system-system communication between networked embedded systems.

¹GreenSocs homepage: www.greensocs.com.

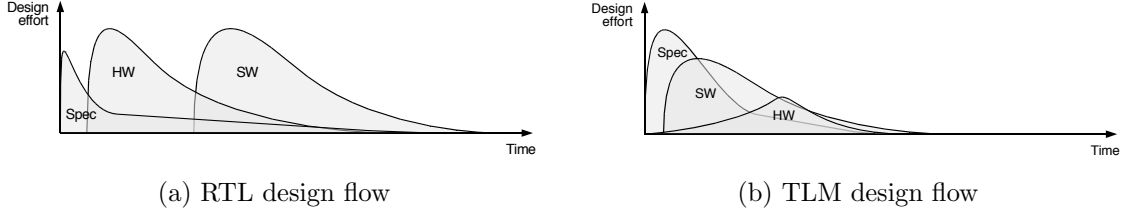


Figure 1.2: Expected advantages of TLM over RTL design

For each of these communications the implementation details differ. While for on-chip communication memory-mapped buses and networks-on-chip are used [137], software-hardware and software-software communication requires device driver and operating system modeling for the selected processor cores [35, 129]. Communication architectures for chip-chip communication (e.g., PCI ExpressTM [105]) differ from architectures for on-chip communication (e.g., AMBATM [5], CoreConnectTM [53]), and the protocols used in interconnects for system-system communication (e.g., EthernetTM [58], USBTM [127]) differ completely from the protocols used in a SoC.

Thus, getting the communication architecture right becomes a primary focus of embedded system design. As system functionality is defined by the inter-operation of the various system components, local communication bottlenecks can have global impact on the overall system performance [63]. In this context, C/C++-based hardware/software co-design languages have found entrance into industry. Here, especially the system description language SystemC has attracted attention and has been standardized by the IEEE in 2006 [60]. In addition to extending C++ with new language elements for describing hardware and software at different levels of abstraction, SystemC inherently supports the concept of *transaction level modeling* (TLM) [18]. TLM allows describing the communication in a system in terms of abstract operations (transactions). This can greatly speed up the design process, as software development can start in parallel with hardware refinement, using an abstract *virtual prototype* of the hardware platform as a base (figure 1.2).

Currently, TLM is not applied in the industry on a large scale, because there is a lack of adequate TLM tools available. According to analysts of the electronic design automation (EDA) industry, this is “the only thing keeping register transfer level (RTL) tool sales growing.” [85]. The reasons are twofold. First, there is no common interoperability standard. A diversity of opinions exists about how TLM models should look like, and each IP vendor promotes another interface standard. Thus, there currently is little chance that TLM models built by others are interoperable with the own. Secondly, abstraction allows a system to be described fast and at a reasonable cost but it also casts a shadow of doubt over the accuracy of performance analysis data gained with such models. The publications on this topic shows a broad range of different, often ambiguous and even incoherent interpretations of TLM abstraction.

This PhD research therefore was motivated by the goal to examine new methodologies for communication modeling with SystemC. Here I focussed on the systematic application of the TLM approach within an IP-based system-level design flow. The GREENBUS extension framework for the SystemC language was developed for

- efficient modeling of SoC using heterogeneous IP at different (mixed) levels of abstraction, and
- high-performance communication architecture exploration with such models.

GREENBUS enables inter-operations between many of the existing communication modeling approaches and performance analysis tools. The goal was to bring the fun and spirit of LEGO to TLM design: allow TLM engineers to always use the modeling techniques and tools that best supports their specific style and their specific needs, so that they can operate effectively and write their models quickly.

1.2 Summary of contributions

The main contributions of this thesis are:

An interoperability standard proposal for transaction level communication modeling using heterogeneous IP at different levels of abstraction, and a generic TLM fabric implementation for SystemC to support it, namely GREENBUS.

A generic TLM router for GREENBUS, which allows for high-speed yet accurate simulation of the state-of-the-art communication architectures (buses, networks-on-chip) at different levels of abstraction.

A model instrumentation framework for performance analysis and debugging of GREENBUS models using industry-standard tools, called GREENCONTROL.

In addition, surrounding concepts and tools for embedded software development and rapid prototyping with GREENBUS are presented.

1.3 Outline of the document

This document splits up into two main parts:

I. Efficient communication modeling with SystemC: the first part concentrates on the first two aspects of this PhD research: what is the *status-quo* in transaction level communication modeling and how should a generic TLM fabric look like to enable systematic yet flexible embedded system design with heterogeneous IP? After an overview on TLM and the related work in chapters 2 and 3, chapter 4 details the generic communication modeling approach and its implementation in the GREENBUS TLM fabric.

II. Architecture exploration and analysis: in the second part, a generic router concept for GREENBUS is presented which enables communication architecture exploration (chapter 5). Several bus protocols and also a network-on-chip (NoC) have been implemented with this approach and I discuss the quality of the simulation results. Chapter 6 presents concepts and implementations of surrounding tools for performance analysis and debugging with GREENBUS.

The second part winds up with a conclusion and an outlook on future work.

The appendixes give an overview on the status-quo in SoC communication architectures (app. A) and provide design patterns for the development of own user application programming interfaces (user API) for GREENBUS (app. B). Appendix C outlines concepts and tools for early software development and rapid FPGA prototyping with GREENBUS, and explores the potentials and limitations of the presented approach with a video processor case study.

Several of the concepts presented in this thesis have become subject to fruitful research cooperations with my colleagues at E.I.S. and the growing GreenSocs community, who in return gave valuable feedback. I usually use first person throughout the document, but for content that has been influenced by such feedback I switch to ‘we’.

The reference implementation of GREENBUS is both available on DVD [67] and for download [68]. The DVD also includes all other tools developed in this thesis.

Trademarks and registered trademarks used in the text (such as GREENBUSTM) are indicated at their first appearance.

2 Basics of Transaction Level Modeling

Contents

2.1 Overview	7
2.2 SystemC	10
2.3 TLM terms and design flow	12

2.1 Overview

Transaction level modeling has been touted to considerably improve productivity in system-on-chip design. Recently many popular SoC development environments have been flavored with the spirit of TLM, typically based on the favorite design language for TLM, which seems to be SystemC. The promise is that with transaction-level prototypes, buses and networks-on-chip (NoC) can be simulated magnitudes faster than with register transfer level (RTL) models, while achieving almost the same accuracy of simulation results, and all this early in the design cycle where decisions made on these results have the biggest impact.

However, despite its power and broad acceptance, transaction level modeling is an ambiguous term. The flurry of publications reveals a variety of interpretations. This chapter starts with an informal overview on the general TLM ideas and their support through the SystemC language. Thereby key terms are used without being fully explained. Then the generally accepted concepts for TLM-driven embedded system design are outlined and in this context the TLM terms which are used throughout this thesis are refined.

2.1.1 Raising abstraction

In general TLM is understood as a modeling technique promising a level of abstraction like RTL but higher, where the key feature is a ‘transaction’. Instead of considering the several data transfers of a communication separately (slow simulation) a transaction encompasses a communication procedure ‘as a whole’ (fast simulation).

TLM models consist of processing elements (PE) that communicate via transactions. The goal is to model data processing (computation) independent from data exchange (communication). This presumes that systems naturally can be decomposed into computation and communication. System-level description languages such as SpecCTM [34] and SystemC [44] support this approach with special concepts for a ‘transaction level description’ of communication, namely ports, interfaces, and channels. This has a number of advantages:

- *Computation and communication can be modeled differently abstract.*

Abstract processing elements can exchange data via an RTL bus model. Likewise, RTL level processing elements can be connected by an abstract transaction level FIFO channel.

- *True hardware-software co-design.*

Abstract processing elements can be integrated to a hardware platform prototype. Such virtual prototypes simulate much faster than RTL models and can be used for early embedded software development. This enables better hardware-software co-design, as the software developers need not wait for the RTL models to be complete.

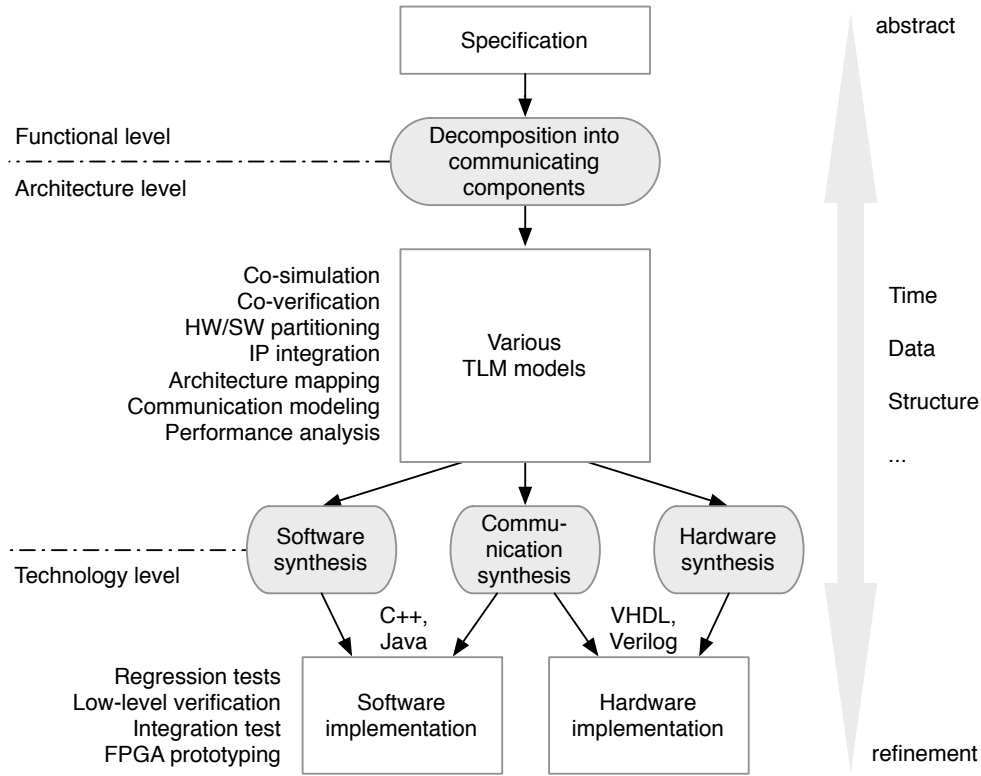


Figure 2.1: Embedded system design with TLM

- *Stepwise refinement and mixed-mode verification.*

An abstract executable specification of the system components and of the overall system architecture can be created, without making any distinction between hardware and software. Refined, less abstract models of the system components can be integrated incrementally. ‘Mixed-mode’ simulation enables verification against the ‘golden model’.

- *Better design space exploration.*

Performance analysis of a given system architecture is enabled in terms of various aspects, including computation, communication, power, real-time, and cost efficiency. Using abstract models, implementation alternatives can be quicker balanced against each other and efficient ‘what-if’ analysis helps to estimate the effects of critical design decisions.

When systematically applied, TLM can support a ‘meet-in-the-middle’ design approach, wherein top-down refinement of application-specific hardware and software is mixed with bottom-up integration of existing IPs. The goal is to decouple the development of the hardware, software, and communication architecture for a new system, such that these tasks can be performed independent from each other, using IPs from different providers. Keutzer et al. refer to this approach with the term *orthogonalization of concerns* [66].

Figure 2.1 outlines embedded system development with TLM and shows the various embedded system design phases that can take place in parallel.

2.1.2 Where is the level in TLM?

With transactions, we can model communication at a higher abstraction than with RTL. But unlike registers, transactions are not well defined. While many feel they have an understanding of the term,

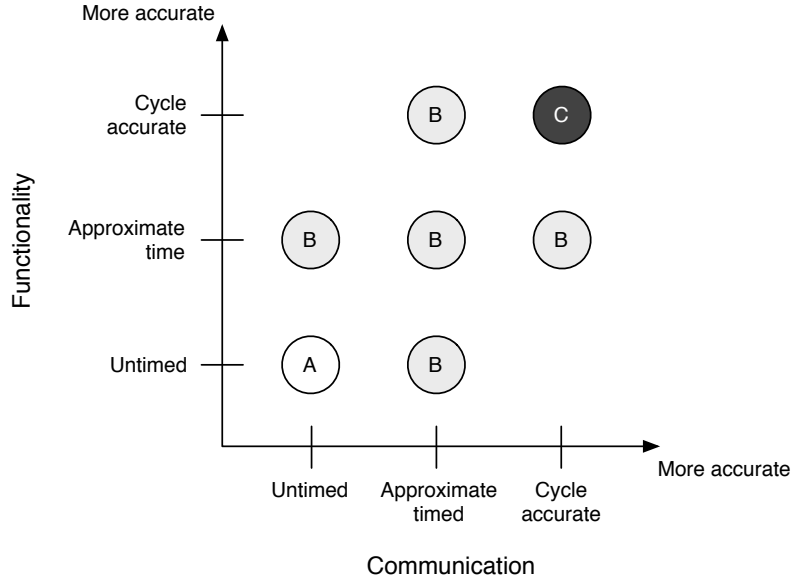


Figure 2.2: TLM abstraction levels in terms of timing accuracy (from [18])

it remains elusive. Furthermore, multiple different levels of abstraction are offered in parallel under the umbrella of the term ‘TLM’.

Figure 2.2 shows some of the abstraction levels we can define when considering only one of the many aspects of a TLM model, its timing accuracy. This approach has been developed by Cai and Gajski in [18]. A TLM design flow would typically start in the lower left corner with an untimed description of both functionality and communication (A). Then, communication and functionality will be incrementally refined, thus adding timing information to both domains (B). Finally, a level of abstraction similar to RTL will be reached (C).

This approach enables the classification of a model in terms of its time *abstraction domain*. Other abstraction domains may be described more abstract or less abstract as well, such as

- *data accuracy* (e.g., abstract data objects usually do not define how the contained information will be organized in physical memory);
- *algorithmic accuracy* (e.g., a functional PE may contain a single thread while its RTL model implements parallel state machines); and
- *structural accuracy* (e.g., with decreased abstraction, high-level components are broken into lower-level subcomponents).

More abstraction domains can be identified. The communication accuracy of a model, for example, mainly depends on its timing fidelity, and may involve some assumptions about data accuracy as well (e.g., technology-specific sideband signals of a physical communication link may be ‘abstracted away’ in higher level models). The computation accuracy of a model may depend on both algorithmic accuracy and data accuracy (e.g., consider fixed point vs. floating point calculation).

Model abstraction therefore is multi-dimensional. In this context, the term ‘level’ is misleading, as it is impossible to unambiguously define a level in a multi-dimensional space. However, to adhere to the related work where the term abstraction level is widely used, we here restrict our view to the communication abstraction domain, which is in the first instance related to the time abstraction domain (fig. 2.2). With the aid of this approach, we will use ‘abstraction level’ as a synonym for ‘communication abstraction level’ in the following.

In conclusion, the level in TLM is a rather ‘thick’ level. The related work shows that the optimal abstraction mix first and foremost depends on the use case, and differs between applications [17, 32, 38, 79, 109]. While for some of the design tasks in fig. 2.1 specific abstraction domains are more important than the others (e.g., time and data accuracy for communication modeling and performance analysis), other design tasks require all abstraction domains of the model to fulfill a specific accuracy (e.g., hardware synthesis).

The term TLM should be seen as a ‘paradigm’ or ‘methodology’. In this thesis we give it a more precise meaning based on the concepts supported by the SystemC language, which are described in the following.

2.2 SystemC

The system-level description language SystemC enables to describe the computation of a system separately from its communication [14, 41, 44, 91]. The underlying language concepts are not exclusively targeting at transaction level modeling – e.g., RTL modeling is supported by SystemC as well – but they are especially suited to support the TLM approach. SystemC is developed since 1999 and in 2006 has been approved as IEEE standard 1666. The current version of the language specification is 2.2 [60]. An open-source simulator for SystemC is provided by the Open SystemC Initiative (OSCI), who also drives the standardization process.

SystemC is based on the C++ language. The IEEE defines it as an “ANSI standard C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are a hybrid between hardware and software.” [60]. Although being just a class library, using SystemC feels like programming in another language. The library makes excessive use of templates, type definitions, operator overloading, and preprocessor macros to add new syntactic features for high-level modeling and hardware development.

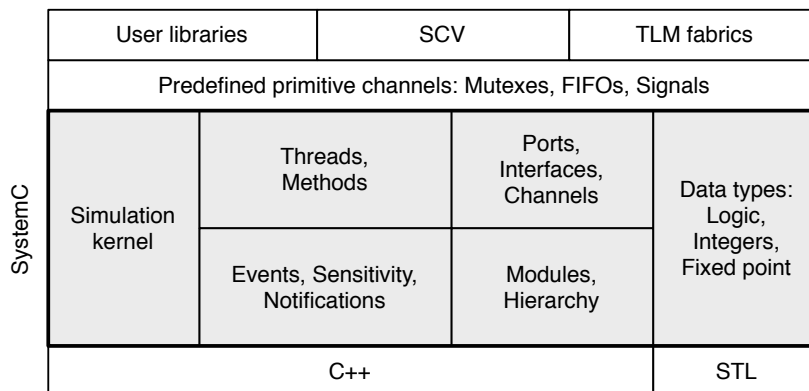


Figure 2.3: Structure of SystemC

Figure 2.3 shows the structure of the SystemC library. The provided core language elements are organized in groups for (from left to right):

- Simulation;
- Concurrency and synchronization;
- Component and communication modeling; and
- Data representation.

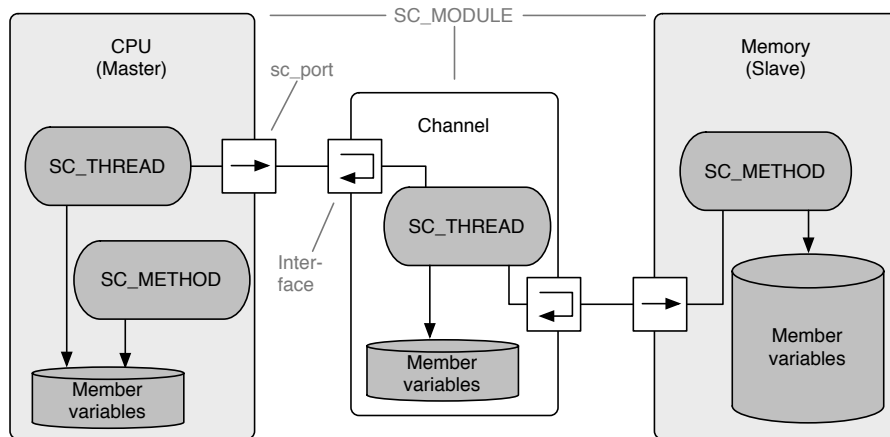


Figure 2.4: SystemC TLM with two PEs connected by a channel

2.2.1 Modules, ports, and channels

Modules (`SC_MODULE`¹) are the basic language construct to describe design structure and hierarchy. Modules contain methods and variables that implement processes. Modules can contain nested modules to build hierarchical structures.

Communication between modules is enabled by *ports*, *interfaces*, and *channels*. Ports (`sc_port`) metaphorically speaking sit at the module boundaries. Each port is associated with an interface (`sc_interface`). It specifies a number of methods that can be called on the port (e.g., `read`, `write`), but does not implement them. The name of a port's interface is given as a template parameter, e.g.:

```
sc_port<sc_signal_in_if>
```

Ports enable modules to make interface methods calls (IMC) on channels. Thus, they act as intermediate in channel access. SystemC does not provide a special construct to model channels. Instead, channels are realized as modules that inherit from and implement one or more interfaces. An illustration of these concepts can be found in figure 2.4.

SystemC provides two techniques to implement processes inside modules:

- `SC_THREAD` processes can run in an endless loop. They can be suspended by calling the SystemC `wait` method. They are resumed after the given time period (e.g., `wait(20, SC_NS);`) or upon an event (e.g., `wait(e);`). In the meantime, other processes can run.
- `SC_METHOD` processes are non-interruptible. They are intended to be executed several times during simulation and therefore can be made sensitive to `sc_events`. Every time an `sc_event` occurs which is on the sensitivity list of an `SC_METHOD`, the method process gets executed. The sensitivity list can be modified during simulation runtime (dynamic sensitivity). Method processes should not contain an endless loop, as this would prevent the execution of other processes.

The execution semantics of `SC_METHOD` processes is close to that of combinatorial logic, whereas `SC_THREAD` processes are meant for behavioral modeling of concurrent processes.

¹Note that most SystemC keywords start with an 'sc.' prefix.

2.2.2 Events and the simulation semantics of SystemC

Events are modeled using the `sc_event` class. Execution of SystemC models is performed by an *event-driven simulator*. Events can be notified either immediately (i.e., the event is fired at the current simulation time) or in the future by specifying a notification delay. For example,

```
e.notify(20, SC_NS);
```

schedules event `e` to be notified in 20 nanoseconds. A special case are zero-time delays, which can be used to model nonblocking assignments².

Explanation 1. The SystemC simulator supports the concepts of *simulation time* and *delta cycles*. It therefore uses two queues, an *event queue* that contains all events that have been scheduled for notification in chronological order, and a *process queue* that contains all processes marked runnable. Initially, all processes are marked runnable and executed once. During simulation, first all immediate events are processed. For each of them, all processes are marked runnable that list this event in their sensitivity list (or wait for it). If no more immediate events are in the queue, runnable processes are executed one after another until the process queue is empty.

Then, simulation time is advanced by one *delta cycle*, so that zero-time events will now be processed. Again, processes are marked runnable and executed. Delta cycles are performed until no more zero-time events are in the event queue (and no new ones have been added). Then *simulation time* is increased, thereby using that amount of time that is specified by the topmost event in the queue. Simulation comes to a halt when both the event queue and the processes queue are empty. This behavior is illustrated in figure 2.5.

Using events, a process can wait for another process to finish some operations. Thus, events enable process synchronization and are the fundamental means for implementing communication channels.

2.3 TLM terms and design flow

With the modeling concepts that SystemC supports we can define a list of TLM terms which will be used throughout this thesis. This chapter also outlines their meaning in a TLM design flow.

2.3.1 Structure of TLM models

The basic building blocks of SystemC models are modules, ports, interfaces, and channels. The latter three terms – port, interface, channel – can be taken as are. Modules, however, require a more precise consideration. SystemC modules can contain methods, processes, variables, ports, and nested modules. They can model both computation and communication. For modules that implement interface methods in order to enable communication we use the term channel as explained above. To clearly distinguish channels from computation related modules, we follow [34] and introduce a further term:

Explanation 2. A module that models *computation* is referred to as *processing element* (PE).

Although this suggests a strict separation of computation and communication, in fact PEs typically may involve some internal communication as well (e.g., shared variables). Likewise, the implementation of the communication mechanisms in a channel requires some computation. But for the purpose of designing embedded system architectures, we think of PEs as models for hardware and software components, whereas their communication is modeled using channels.

Explanation 3. A single PE or channel already is a *TLM model*. By hierarchically connecting PEs and channels via ports, larger TLM models can be constructed.

²For a discussion of nonblocking assignments, see [28].

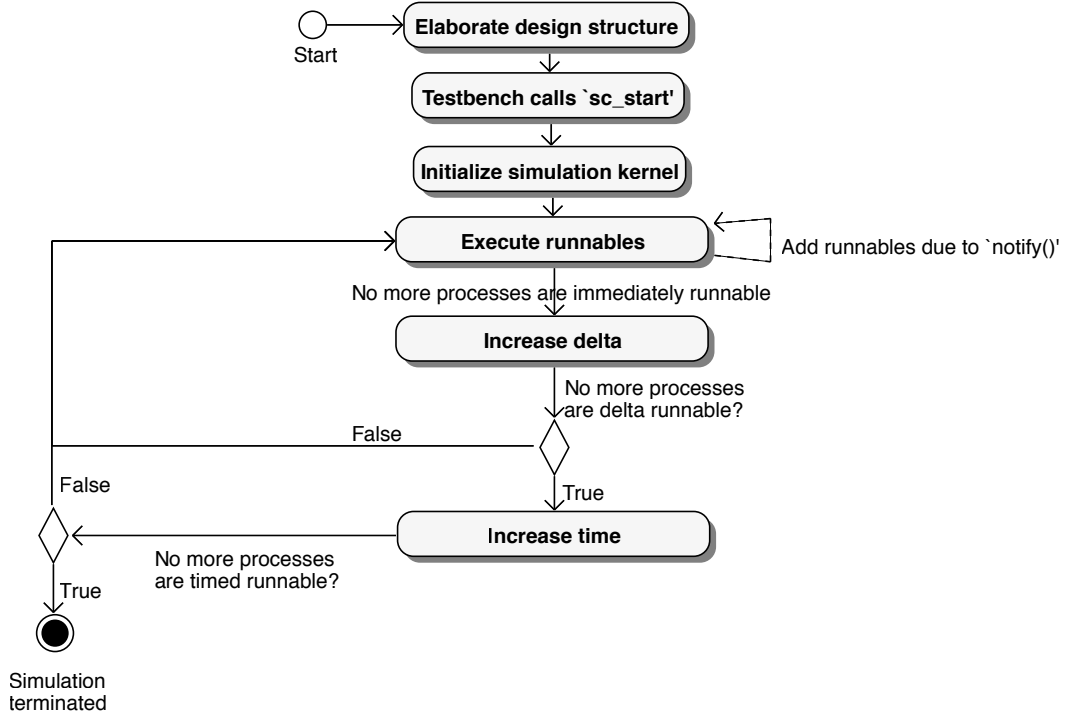


Figure 2.5: SystemC event-driven simulator

The concept of PEs, ports, and channels is illustrated in figure 2.6. It enables object-oriented communication modeling at different levels of abstraction and helps to separate communication and computation in the source code.

2.3.1.1 Separation of behavior and architecture

As already mentioned, PEs and channels may have different abstractions. While some PEs and channels in a TLM model may be high-level functional models, others may be described using a less abstract model of computation (MoC) such as RTL, finite state machines, etc. [62]

Using abstraction, behavior can be described independent from architecture. For example, the abstract communication implemented by a FIFO channel (with simple `put` and `get` interface methods) need not necessarily reflect the complex communication semantics of a shared-memory bus architecture onto which the FIFO channel may be mapped later in the design process. Likewise an abstract PE may provide the functionality of a JPEG encoder, using a totally different MoC (e.g., Kahn process networks [44]) than an RTL model of the same JPEG encoder.

2.3.1.2 System-level design

Abstract behavioral models support fast design capture and attain high simulation speed. In a typical system-level design flow, the engineer will start with a composition of abstract behavioral PEs and channels, with the goal to stepwise refine them towards an architectural realization in terms of hardware, software, and communication architectures. During this refinement process, for each PE it is decided whether it is to become hardware or software. The appropriate model of computation for the refined implementation of the PEs is chosen accordingly. The source code refinement adds more and more details in terms of data, time, and algorithmic accuracy. Likewise, the channels are refined, which may but need not necessarily take place in parallel with PE refinement (cp. fig. 2.1).

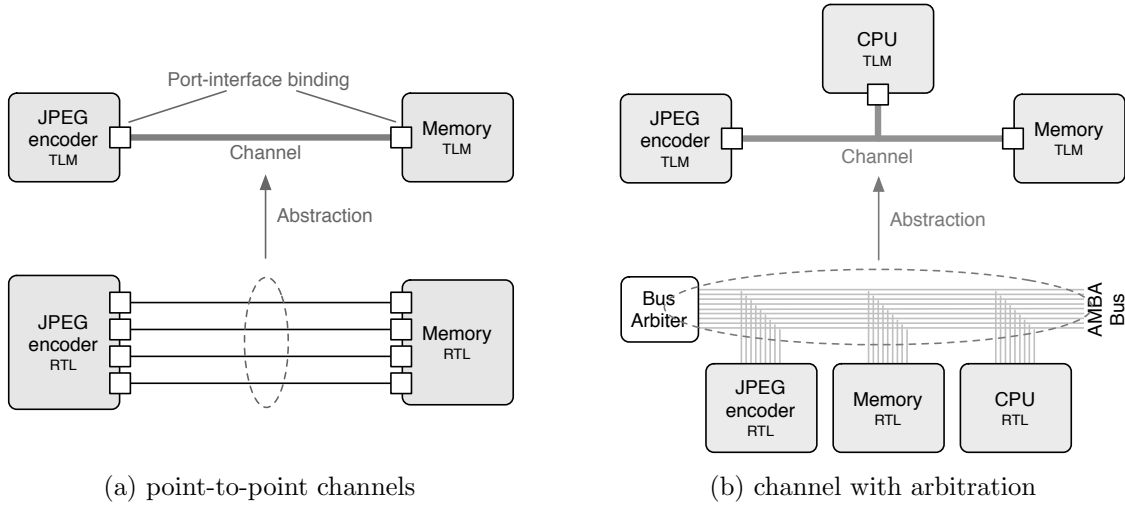


Figure 2.6: Abstract channels may, but need not necessarily reflect the behavior of a physical communication architecture

2.3.1.3 Architecture mapping

When a suitable hardware/software partitioning has been found, the PEs and channels must be mapped onto a target architecture. PEs that are to become hardware can either be manually refined towards an RTL representation (which then can be fed into a hardware synthesis tool), or they can be mapped onto existing hardware IP. Channels can be mapped onto existing physical communication architectures, e.g. a shared-memory AMBA bus such as shown in fig. 2.6b. Although on the first glimpse software PEs written in SystemC may be directly executable on a target processor, they typically also need to be refined. They need to be adapted for the target operating system and their communication with the hardware requires the incorporation of device drivers.

2.3.1.4 Mixed-mode verification and design space exploration

Successive simulation of the refined models allows for functional verification against an initial golden model. At each phase of the design flow the designer should be able to validate the correctness and quality of his models. Furthermore, the designer should be able to quickly explore the trade-off in terms of critical design constraints for different architecture mappings. This presumes that heterogeneous PEs and channels, as well as existing RTL IPs and abstract variants of them can be arbitrarily composed, independent of their model of computation and level of abstraction. For example, the designer should be able to replace an abstract channel with a less abstract model of an on-chip bus like AMBA, and if the simulation results do not match the constraints (e.g. low communication performance), replace it by a model of a completely different communication architecture, e.g. a network-on-chip. In a similar way the engineer should be able to explore different mappings of abstract PEs onto existing IPs.

2.3.2 The various roles of channels

Channels therefore are a key element in the TLM methodology. They must enable interoperation of high-level and low-level PEs so that the designer is free to always use the best compromise in the design cycle. Moreover, they must enable design space exploration. Especially, they should allow to compare different communication architectures for the system.

Explanation 4. While the syntax of the interface methods of a channel is defined by its interface(s), their behavior depends on the actual implementation. The combination of interfaces and channel be-

havior describes a communication *protocol*. In communication technology a variety of protocols and approaches to describe them exist [80]. In the context of SystemC channels, the protocol implemented by a channel specifies a number of rules which determine the allowed arguments and invocation sequences for the interface methods in order to send information over the channel. This also may include timing rules.

2.3.2.1 Abstract channels vs. architecture mapping

When starting with a TLM design flow, often abstract protocols such as synchronous message passing or FIFO channels are used [44, 73, 84, 109]. They relieve the designer of coping with low-level communication details and let him concentrate on the system functionality (cp. the ‘functional level’ in fig. 2.1). For example, the abstract point-to-point channel in figure 2.6a may provide two object-oriented interface methods, `read(img&)` and `write(img&)`, with which the JPEG encoder can read an abstract image object from the memory and write back a JPEG-compressed version of the image, each with a single interface method call.

In contrast, after architecture mapping has been finished, channels that model physical communication architectures are used. They implement the low-level interfaces to which the PEs have been refined in order to enable automated synthesis (cp. the ‘technology level’ in fig. 2.1). For example, the RTL channels in the low-level model in fig. 2.6a transfer only one data primitive at a time (e.g. `int`, `byte`, ...) and implement the timing protocol of clocked register transfers. The `sc_signal` channel of SystemC provides this functionality.

2.3.2.2 Communication architecture exploration

While both channel types are suitable for their respective use cases, they do not support the design tasks in the ‘architecture level’ block in fig. 2.1. In particular for architecture exploration a channel type is required that is a ‘hybrid’ of the above described types:

1. it must enable the simulation of a specific physical communication architecture such as the bus in fig. 2.6b;
2. it must allow the connection of PEs with interfaces that differ from the ‘native’ interface of the simulated communication architecture.

With traditional RTL models only the first requirement can be met. If `sc_signal` channels are used to model a bus as in fig. 2.6b, the result will be a very close reproduction of its wires, registers, and their logical interconnections. In order to connect a PE to such a bus model, a multitude of ports is required to bind them to all the bus signals, and a simple data transfer such as a burst write involves reading and writing dozens of signal values in accordance with the channel protocol. This prevents efficient design space exploration, as switching to another bus model requires the re-implementation of large portions of the PE source code. Moreover, the simulation performance of such bus models is unsatisfying due to their low abstraction.

An approach to meet both requirements and attain higher simulation performance is to create a channel that implements an abstract interface but still supports architecture exploration.

Explanation 5. With the term *communication architecture functional model* (CAFM), we denote special channels that indent to simulate critical characteristics of a physical communication architecture sufficiently accurate while however being more abstract than an RTL model. The goal is to achieve higher simulation speed and provide more abstract interfaces in order to enable faster design space exploration.

For example, the abstract bus channel in fig. 2.6b may provide object-oriented `read` and `write` interface methods to the PEs, whereas internally an AMBA bus specific serial data transmission is implemented in order to simulate the special AMBA protocol timing characteristics.

CAFM address the ‘logical’ communication architectures typically used in today’s embedded systems. In most cases being shared buses, they are specified at the RTL level of abstraction and encompass a number of RTL signals which are evaluated synchronously to a clock signal. The common goal in CAFM design is to simulate its related RTL model at a level of abstraction higher than RTL which meets both of the following two requirements:

1. the CAFM is able to precisely estimate which PE may access the modeled communication architecture at which point of time and how long the transactions last, in terms of clock cycles.
2. the CAFM disregards as much details of the simulated RTL model as possible in order to achieve higher simulation performance.

In other words, a CAFM provides a means to perform transactions among PEs and is able to calculate the time points in terms of a clock signal when these transactions would start and end if they were performed over the related RTL model. The timing fidelity provided by CAFMs may vary. While information about the start and end time points of transactions may be sufficient for high-level architecture exploration, lower level models may demand for information about certain intermediate time points as well.

To be applicable with SystemC, a CAFM must be able to synchronize its timing with the SystemC simulation time, and it must provide a proper SystemC representation of the control and user data in the simulated RTL model.

2.3.2.3 The TLM interoperability gap

Several approaches to CAFMs have been presented in the related work. They will be discussed in chapter 3. There exists a broad range of proposals on how CAFMs should be designed, and each of them proposes another coding style which results in different kinds of abstraction and different interfaces, most of them being incompatible.

For the designer this means that though he has a lot of CAFMs available, he cannot make use of them for architecture exploration. The interfaces used are CAFM specific. PEs with another interface cannot be directly connected, as they are not interoperable with the CAFM.

The commonly proposed solution to overcome this ‘interoperability gap’ is to make use of adapters.

Explanation 6. *Adapters* are special channels that implement two different interfaces. Interface method calls on the one interface are mapped onto appropriate interface method calls on the other interface, thereby translating one protocol into another.

Example 1. Consider figure 2.7. It shows an example of an MP3 player system comprised of four PEs. To simulate its communication bus, a CAFM is used, here of the STBus architecture from ST Microsystems. The interface used by ST for their CAFMs is called TAC (transaction accurate communication) [41, 119]. The memory and the software model have been designed to call TAC interface methods and thus can be directly connected to the CAFM. The audio controller and the MP3 decoder, however, require adapters.

Adapters can be developed for a variety of applications. They enable to connect PEs at different levels of abstractions and PEs that use different interfaces. Unfortunately, no generally applicable methodology exists to generate adapters automatically. Hence, with the existing CAFM approaches, new adapters need to be developed whenever

1. the designer wants to explore another communication architecture and its CAFM has another interface;
2. the designer replaces one of the PEs with another version which has been developed for another interface.

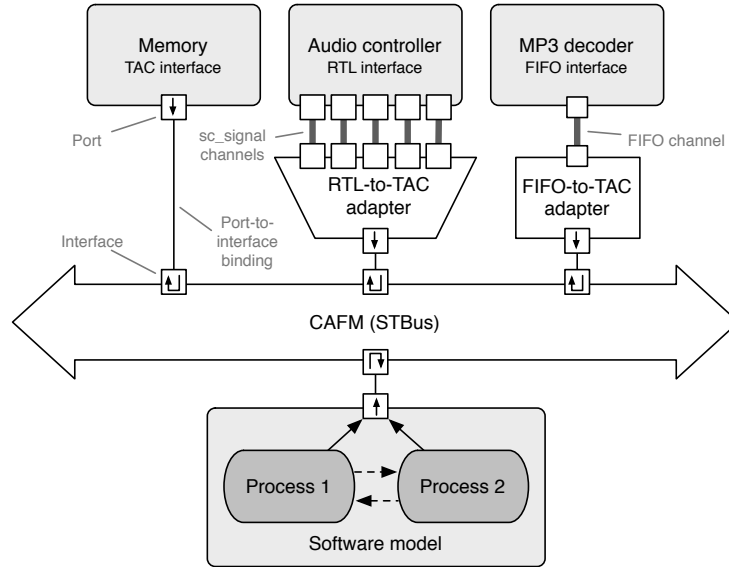


Figure 2.7: Example TLM with heterogeneous PEs connected to a CAFM

2.3.3 On the notion of transactions

With the above considered terms we now are able to describe the structure of TLM models. We also can make some statement about its abstraction level (though the latter requires further analysis). But we still lack a clear understanding of the linchpin of TLM: the *transaction*.

In a totally general sense, any kind of data transfer between PEs can be understood as a transaction. There is no common definition of the extent of a transaction. Thus, when designing a custom abstract channel, the designer is free to determine whether a transaction over his channel is defined as a single interface method call (e.g., **write**), or as a specific sequence of interface method calls (e.g., **request** → **response**). Likewise, in one model transactions may terminate after a given period of time, while in another model they may last for an indefinite amount of time. The notion of transaction therefore is associated with several aspects of a channel in parallel, including its protocol, its abstraction, and its intended use case. We cannot give a universal definition for the term.

In the more specific context of CAFMs, however, we can harness the fact that CAFMs model the specific protocol of an existing physical communication architecture. Here, we can develop a generic notion of transaction for CAFMs based on the properties of their protocols. We assume that in all protocols used in today's embedded system and system-on-chip architectures two fundamental states of communication can be identified:

1. *idle* – no communication takes place at the moment, a new communication can begin;
2. *busy* – there is an ongoing data transfer, other communication requests must wait until the transfer has ended.

Depending on the actual protocol, each of these two states may include a multitude of sub-states, and there may be several options how these sub-states can be traversed. A common approach to describing protocols in a formal manner are finite state machines (FSM) [80]. The more complex a protocol is, the more different states it may go through during a busy phase. But there always is a start and an end state, and a finite set of states in-between. We can analyze a protocol description with regard to these criteria and from this, in the context of a CAFM for this protocol, can define its notion of transactions as the sequences of interface method calls that change the communication state of the simulated protocol from idle to busy and to idle again.

2.3.4 Conclusion

Transaction level modeling is a powerful methodology for the design of complex hardware-software systems. The object-oriented separation of computation and communication modeling, and the ability to mix models at different levels of abstraction gives great flexibility in the design flow, enables higher simulation speeds and contributes to IP-based design.

While the structure of TLM models is generally defined, there is a lack of clarity in terms of communication modeling. In particular, models of different designers cannot interoperate without tailor-made adapters, because their communication abstractions are not clearly defined and their notion of transactions differ. In the context of communication architecture exploration, an approach is to develop a generic, architecture-independent notion of transaction, onto which different physical communication architecture protocols can be mapped. In the following, the potentials of this approach to develop a generic TLM communication modeling fabric for SystemC will be examined.

3 Towards a Generic TLM Fabric for SystemC and Related Work

Contents

3.1	Motivation	19
3.2	Communication modeling approaches for SystemC	21
3.3	Survey of TLM frameworks	26
3.4	Discussion	28

3.1 Motivation

Figure 3.1 shows a block diagram of the OMAPTM-3 SoC architecture from Texas Instruments¹, which has been introduced in February 2007. OMAP 3 SoCs are comprised of various hardware components arranged around an ARM RISC processor core. Several digital signal processors allow for video, digital TV, and audio processing, thus empowering designers to build the next generation of internet-mp3-digicam-video-hybrid cellphones. Depending on their custom-tailored configuration, OMAP-3 SoCs reach a complexity of up to 90 million transistors [123].

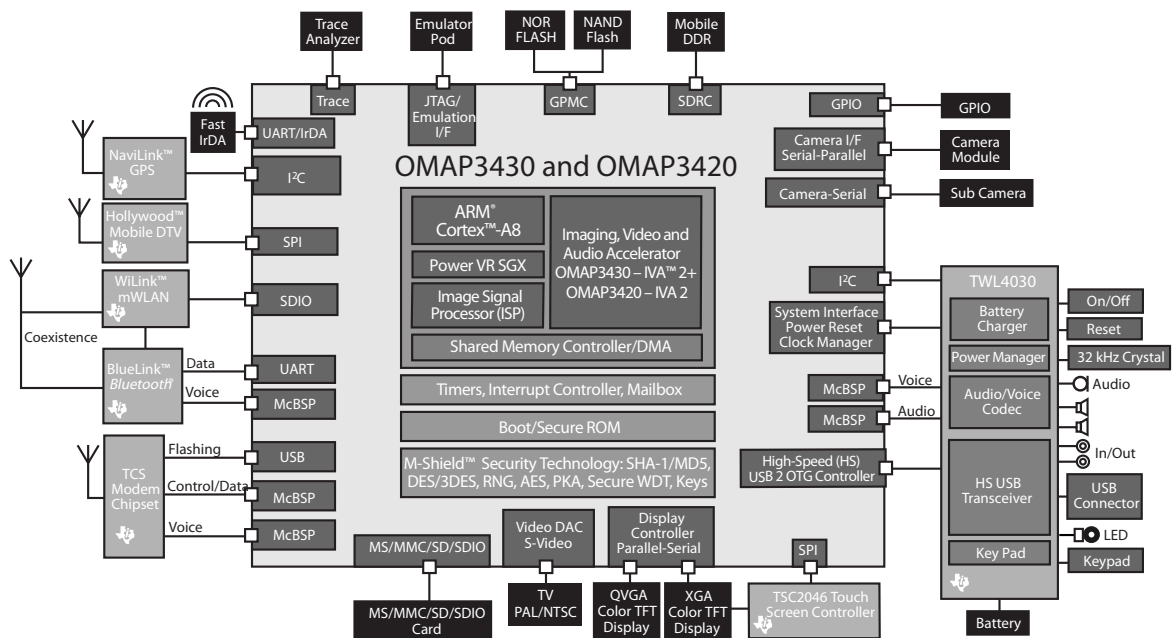


Figure 3.1: Texas Instruments OMAP-3 system-on-chip (from [126])

¹OMAP stands for Open Multimedia Applications Protocol and is a trademark of Texas Instruments.

As typical to today’s SoC platforms, OMAP-3 implements a component-based design approach [21–23]. That is, all hardware components are available as ‘soft’ IP cores. A development tool allows to assemble customized OMAP-3 SoCs from this IP library.

Platform-based design approach

Other SoCs come with complex block diagrams too, but I have chosen the OMAP-3 architecture because it highlights three important trends. First, it comprises *heterogeneous* IP cores. Their source code has been developed by a variety of companies. Second, OMAP-3 is not a single SoC but a *SoC platform*, allowing for assembly of tailor-made chips for different customers. Third, OMAP-3 is one of the industry’s first SoC platforms which comes complete with a fully-fledged *system-level design environment*. SystemC performance models are provided for all IP cores of the platform [3]. They enable to build TLM models of a customized OMAP-3 platform easily, boasting high-speed functional simulation for fast architecture exploration and early software development.

While this example highlights the power of SystemC and TLM, a second look at OMAP-3 also reveals the unsolved deficits of the methodology: OMAP-3 virtual prototyping with SystemC would not work out so smoothly if Texas Instruments hadn’t optimized all their models for interoperability. All OMAP-3 IP cores are available as both TLM PEs and as synthesizable RTL IP for a specific target platform. All PEs use the Open Core Protocol (OCPTM [94]) as standardized communication interface and share the same level of abstraction. Thus, they fit together seamlessly.

Other recent examples of such platform-based SoC architectures are ST NomadikTM [120] and NXP/Philips NexperiaTM [93]. This *platform-based design* approach has been promoted by Sangiovanni-Vincentelli et al. since 2000 [29, 30, 66, 111]. Based on dedicated, application-specific standards, a library of interoperable system-level models is developed for a specific hardware platform.

Unsolved problems and motivation of this work

However, in most cases such a platform library will not be available. Rather, designing a novel product typically starts with a mixture of heterogeneous models, including established IP from previous products, newly self-developed models, as well as models from third party vendors. If the used interfaces, abstraction levels, and modeling techniques differ, models cannot be integrated efficiently.

Various TLM frameworks, i.e. TLM ‘convenience’ modeling frameworks, have been presented to overcome these issues [10, 19, 25, 27, 31, 39, 42, 79, 95, 103, 109, 112, 113, 116, 119]. The common goal for the TLM frameworks that are being proposed is to ease the construction of model-to-model communication. TLM frameworks typically provide a set of library functions that enable a specific communication modeling style. The communication mechanism is seen as being relatively complex, common to many models, and above all, if it were common, then reuse might be possible.

However, as the following reviews will show, interoperability is mostly disregarded by the existing TLM frameworks. Instead, each TLM framework promotes another proprietary communication modeling approach, mostly with respect to a specific use case or communication architecture it is designed for.

The motivation of this work is to examine whether a TLM interoperability standard reaching across individual protocols and modeling styles can be developed, with the goal to build a generic TLM fabric, which

- supports multiple levels of TLM abstraction;
- enables inter-operation between models of different levels of abstraction (mixed-mode), and models with different interfaces (heterogeneous components), with as little overhead as possible;
- enables to build CAFMs for various bus and NoC communication architectures;
- attains highest possible simulation performance;
- adheres to common standards such as OSCI-TLM [109] and OCP [94].

To the best of my knowledge such a TLM fabric does not exist.

3.2 Communication modeling approaches for SystemC

TLM designers are spoilt for choice. A wealth of TLM frameworks are offered with which to model channels and CAFMs, and each of them boasts another set of features and limitations. In an attempt to get a general idea of the status quo in transaction level modeling, I consider three main aspects of TLM fabrics:

- The API (‘user view’)
- Supported levels of communication abstraction (‘TLM view’)
- The techniques used for the implementation (‘technical view’)

In the following, these three ‘views’ on TLM frameworks are discussed and major differences in transaction level modeling are pointed out. From this list of key issues I think standards in TLM should emerge.

3.2.1 User view

The intention of transaction level communication modeling is twofold:

1. to abstract from hardware-dependent details so that the communication model can be reused with different hardware platforms and the designer can (first of all) concentrate on implementing the functionality of his system – for such models, we introduced the general term channel;
2. to create abstract communication architecture models that simulate RTL communication models (mostly of buses) and thereby achieve higher simulation speeds than the RTL models while yet being accurate enough for architecture exploration and mixed-mode simulation – for such special channels, we introduced the term CAFM.

TLM frameworks intend to extend the used design language (SystemC) with new constructs that aim to ease channel and/or CAFM development. From the user’s point of view, the API (presumably) is the most important aspect of a TLM framework. Its basic task is to provide a convenient interface with which to perform transactions over the TLM interconnects.

Explanation 7. Reviews show that there is industry wide cohesion, with transactions being either read or write transfers initiated by a *master*² and processed by a *slave*² [16]. The terms master and slave are used to indicate the direction of the control flow for the transaction, which is from master to slave. The data, however, may flow into both directions.

This is a typical property of shared-memory buses. However, while the roles of masters and slaves are commonly agreed, the interface method calls required to set up and perform transactions differ significantly.

ST Microelectronics’ TAC package for example ([41, 119]), which is based on OSCT’s TLM kit 1.0 for SystemC ([109]), provides simple read and write methods that transport arbitrary data types over point-to-point channels and abstract bus models. TAC operations are blocking and they are not designed to support advanced control of communication behavior such as specification of bus access priorities or byte-enables. Thus, TAC is a good choice for high-level models aiming at early embedded software development, but it does not support communication refinement towards lower levels of abstraction, thus architecture exploration with TAC is limited.

IBM’s CoreConnect models for SystemC ([57]), on the other hand, provide a precise simulation environment for CoreConnect bus architectures. To this end, a tailor-made CoreConnect API with a

²In the following, I use the terms master and slave in replacement for PE where its (temporal) role in terms of a transaction shall be highlighted.

comprehensive set of communication methods has been developed. This API perfectly assists designers who deal with CoreConnect bus interfaces everyday, but is inconvenient for others. Thus, IBM's TLM framework is an excellent tool to create simulation models of already existent CoreConnect chips, but is less appropriate for top-down SoC design.

The two examples show that ideally the API is completely decoupled from the underlying TLM fabric. If, e.g., the IBM models were equipped with a TAC API, that would allow TAC-based software models to communicate with CoreConnect IP over a precisely simulated bus architecture. Put another way, ideally, a designer should be at liberty to choose which interface suits his needs best, independently of the technology and details of the communication mechanism. A designer used to 'speaking' CoreConnect would rather use the CoreConnect API, while a different designer may prefer TAC.

This sounds idyllic, and if the underlying technology is insufficient to support the API it is unrealistic, but it remains one of the key areas in which TLM frameworks can help increase engineers productivity, and get models out quicker. If TLM is to live up to the hype, then one key is providing the TLM engineer with a choice of environments that they can choose from in order to better support their specific style and their specific needs, in which they can operate effectively and write their models quickly. If the API is independent from the communication fabric, then theoretically, swapping out one communication fabric for another should be possible, models should become interoperable. These issues will be revisited below.

3.2.2 TLM view

3.2.2.1 Abstraction

As has been touched upon already in chapter 2.1.2, communication abstraction is closely related to both the time and data abstraction domains. In figure 3.2, two fictitious bus transactions are shown as an RTL wave diagram. A master sends data to a slave. The transactions are initiated by a clocked request signal. After the bus arbiter grants access, the master sets the target address, which is acknowledged by the slave. Then data transmission is performed. The first transaction is a single-beat transfer, whereas the second transaction is a burst transfer. The extent of the transactions is defined by the idle/busy state changes of the protocol (cp. chapter 2.3.3).

Now we can consider different time points of the communication to model it at different levels of time abstraction:

None

No attempt is made to record any of the time points whatsoever. Complete data records can be transmitted by single method calls and simulation speed is very high, which is convenient to software developers. There is a problem with ensuring that multiple masters (i.e. concurrent processes) in a simulation progress at realistic rates and one is not blocked waiting for another. This is typically termed system synchronization. It can be implemented in two ways, either by keeping a very rough estimate of time and ensuring that each master 'yields' after some time to the others.

Or specific 'synchronization points' can be coded into the model (cp. [41, chapter 2.3]). These are points at which synchronization is required, and hence it's only here that other master need to be given the chance to execute. In real-time operating systems (RTOS), this is supported by message queues, semaphores, and special events [40, 83, 139]. The former method has the advantage of being simple to code, but it is always sub-optimal, requires a notion of time (which is otherwise meaningless in an untimed model), and removes the need for the system designer to think about where synchronization happens in the system. However, it is a good idea to categorical avoid such a modeling style, because it often will result in models that boast non-deterministic behavior.

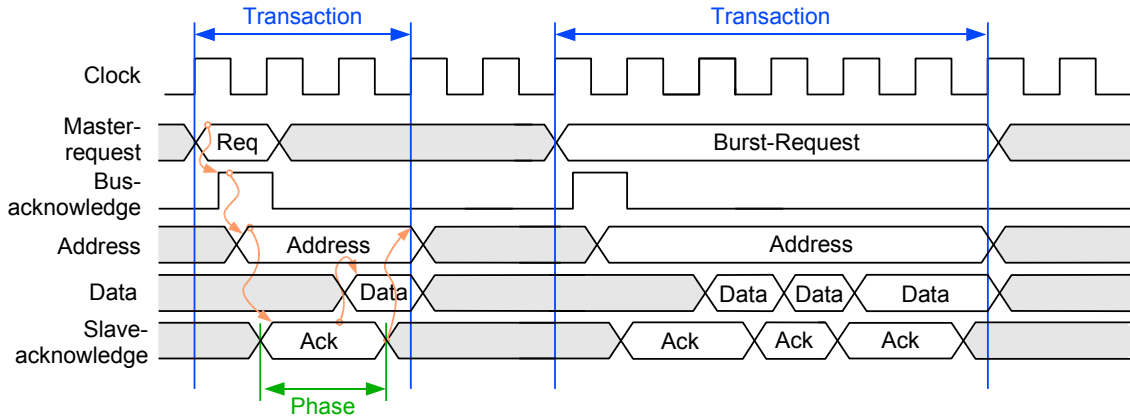


Figure 3.2: Bus Transactions

Transactions

Considering transactions ‘as a whole’ often is referred to as *programmers view* (PV) [41, 109, 119]. PV transaction modeling allows for describing synchronized communication between both hardware and software processes. Here, lightweight interfaces are used to describe PE-to-PE communication with simple abstract methods, such as provided by ST’s TAC package or the FIFO channels that are part of OSCI’s TLM framework.

As only the transaction boundaries are considered, the typical modeling style are *blocking* interface methods. They do not return until the transaction has been completed. The resulting communication model may be either *timed* or *untimed*. If all interface method calls return in the same simulation time point, the model is referred to as an untimed PV model. If some interface method calls let pass simulation time (e.g., one of the communicating PEs calls `wait` to implement a rough time calculation), the model is referred to as a timed PV model. For the latter, sometimes also the term PV+T (programmers view with time) is used.

PV models are proposed as an appropriate entry point for embedded software development. Many TLM hardware/software co-design methodology proposals adhere to PV(+T) communication as standard base for design entry [22, 24, 37, 41, 42, 79, 81, 101, 138]. However, the timing fidelity of PV models is not sufficient for communication architecture exploration. Dynamic delays that occur due to access collisions when multiple masters share a single channel are not taken into account.

Protocol phases

Here it is assumed that the modeled communication can be described as a sequence of protocol phases. Typical protocol phases in bus transactions are init, data transmission, and finalize phases. In terms of the FSM approach to protocol modeling (cp. 2.3.3), a protocol phase can be understood as a sequence of states that must be traversed in order to exercise a specific protocol operation (get bus access, send command, read data, write data, etc.).

A protocol phase encompasses one or any number of clock cycles (limited by the duration of the transaction) and has a start and an end time point. The latter typically are associated with specific signal value changes in the corresponding RTL model, as can be seen in fig. 3.2. These and all signal value changes that take place in-between form the information content of the phase.

The duration of a phase (in clock cycles) is generally determined by the control signals of the related RTL communication architecture model (protocol-specific fixed delays, dynamic delays due to multiple access arbitration, dynamic response times of the PEs). When adequately applied, the phase-based modeling style can provide a ‘cycle-count accurate at transaction boundaries’ (*CCATB*) simulation of the related RTL model. That is, the start and end times of transactions (‘transaction boundaries’)

are exactly estimated. The timing fidelity for the intermediate activities, however, is less accurate. Pasricha et al. show in their work on this modeling style that they were able to create CCATB CAFMs of the AMBA buses with SystemC [103]. The authors assume that the CCATB approach is sufficient for other CAFMs as well, but do not provide solutions. The implementation technique used is not explained and the achieved simulation performance is rather unsatisfying (55% faster than the RTL models). The latter, however, may be a result of the utilized simulation environment.

In contrast to PV models, in a phase-based model the effects of shared medium access contention can be taken into account. Since arbitration of request phases can be considered independently of the other communication phases, e.g., a high priority master can cut off a lower priority transaction (which may have already started). Protocol phases may overlap in time. Hence, phase-based models should use *nonblocking* interface methods. That is, the interface method call to initiate a protocol phase returns immediately, although the phase may be still active. In SystemC models, the end of the phase can be notified either with an event or with another interface method call. This concept for example is used by the Open Core Protocol channel library for SystemC ([95]). Two abstraction levels, OCP-tl2 and OCP-tl1, are provided using the phase approach. At OCP-tl1, a transaction comprises of a request phase followed by several data phases, each transferring an individual data word. At OCP-tl2, all data transfers are merged into a single data phase, transferring a vector of words.

The phase-based approach seems to be a very promising approach to CAFM design, as it can provide satisfying timing fidelity for communication architecture exploration while nevertheless ‘abstracting away’ most details of the related RTL model. However, up to now only very few of the existing CAFMs make use of this approach. A reason may be that there is no SystemC extension or library available to systematically support phase-based CAFM design. Hence, the GREENBUS framework presented in this work focuses on providing such an extension, a generic TLM fabric for phase-based CAFM development with SystemC. It will be shown that GREENBUS not only allows for development of fast CCATB CAFMs, but also provides the possibility to develop CAFMs that support a greater range of timing fidelity grades, from PV over CCATB to cycle-count accurate.

It is important to note that the simulation accuracy of phase-based models strongly depends on the careful selection of the communication time points that are being represented by its phases. The specification documents provided for the buses used in today’s embedded systems (see chapter 5 and appendix A) are rather informal; in most cases the communication rules that make up the protocol are presented in an implicit manner based on exemplary waveform diagrams [6–8, 49, 54–56, 105, 108, 127]. Thus, there is no automatic way to derive CCATB-sufficient phase specifications from the existing protocol descriptions. Nonetheless, my experiments have shown that a straightforward approach is to mainly consider the arbitration signals of the bus (i.e. those signals that are connected to the arbiter). They determine both the request phases supported by the protocol and its finalize phases. The data transfer operations are typically controlled by few control signals.

A review of bus protocols shows that the supported communications can be mapped onto one or more of the following generic phase-based models:

- *Single beat*: The transaction splits up into a request and a data phase, transferring a single data word. The size of the data word is determined by the physical bit-width of the data bus. In some buses request and data phase overlap (i.e. the data signals are set in the same time point as the arbitration signals).
- *Single-request multiple data (SRMD)*: The transaction splits up into one request and several data phases. With each data phase, one data word is transferred.
- *Multiple-request multiple data (MRMD)*: The transaction splits up into a sequence of alternating request and data phases. The resulting communication is similar to a sequence of single beat transactions, but bus access is granted to a single PE during the whole MRMD transaction, which hence (usually) cannot be interrupted.

For the prevalent on-chip bus architectures CoreConnect and AMBA this is exemplified in chapter 4.4.1. For the inter-chip bus architectures PCI Express and CAN (which are quite different), this has been confirmed as well [76, 114] (PCI Express see also 5.4.2).

For point-to-point links and NoCs constructed therewith, we can state that they generally can be modeled with the phase-based approach as well: the protocols used for data transfers between the switches that make up a NoC are typically simpler than those of buses, as they do not need to deal with more than one master at once. In [46], this has been confirmed for an OCP-based NoC (see also 5.4.3). In the related work, even PV models are considered sufficient for NoC modeling [12, 25, 39, 78], as here the main challenge is modeling the switches, not the channels. However, NoCs are still a young and mostly academic research area [64, 137] and buses remain the most often used communication architecture in embedded systems. Hence the concepts presented in this thesis focus on bus modeling, but where applicable NoC modeling is taken into account as well.

Clock cycles

The highest resolution in terms of communication timing is provided by cycle accurate models. Here, the signal value changes at each individual clock cycle in the corresponding RTL model are considered. This enables precise simulations of any interconnect, provided that its communication is synchronized to a clock. Nevertheless, cycle accurate models may perform faster than native RTL models, mostly because the data and functionality used is typically abstracted to being more suitable to execute on a host platform, rather than being an accurate reflection of the hardware implementation.

3.2.2.2 Ambiguity and incompatibility

Interpretation of communication abstraction is ambiguous in the TLM frameworks I have reviewed. For example, the OSSS channels from OFFIS Oldenburg ([42]) in principle allow for cycle accurate simulation of bus architectures, but the provided bus models do not support combinatorial arbitration of concurrent request.

OSCI's new TLM kit 2.0 (which at the time of writing this document is available for public beta review, see [90, 98]) basically supports both PV and CCATB CAFM development, but it is unclear how they should be implemented. Some PV request and response data structures are defined, while those for timing are left to the user to determine (though the expectation seems to be that the PV level request and response structures would be re-used).

IBM's CoreConnect models only support a clocked mode of operation, but from the documentation it does not become clear how accurate these models are.

The OCP channel library provides the most comprehensive coverage of communication abstraction levels. However, the adapters by which OCP channels of different abstraction levels are connected are not made public, and different OCP configurations may result in incompatible behavior.

The plurality of approaches results in the obvious problem that TLM models of different designers are not compatible. Moreover, even models created with the same TLM framework may not fit together, because the designers used incompatible configurations or different interpretations of abstraction.

The whole extent of the problem becomes apparent when we take data abstraction into account. There is no standard for the representation of data in transactions. Many TLM frameworks allow for individual configuration of the transported data types by means of template parameters. Other frameworks use fixed data types in their interfaces. As a result, hoping for interoperability of transaction level models designed by different companies today unfortunately seems futile.

3.2.3 Technical view

Many techniques for the implementation of transaction level communication have been proposed. In general, all TLM frameworks I have reviewed try to make the most of the features of the underlying system-level design language, that is SystemC. While first communication architecture models

based on the PE-port-channel approach (such as the Simplebus example that is provided with the open-source SystemC kernel [61]) were of quite simple nature, today's sophisticated bus CAFMs are comprised of dozens of interacting classes that are often built of complex and interwoven class hierarchies. Features include comprehensive transaction monitoring and debugging, global memory and register models, system-wide address space management, as well as extensive configuration capabilities. Thus, the original basic ports and interfaces are snowed under with extensions and modifications, of which makes clear why compatibility of different TLM frameworks is so hard to achieve.

The specific implementation techniques used in the various existing TLM framework approaches are surveyed in the following.

3.3 Survey of TLM frameworks

To complete the presentation of the related work, this section summarizes the characteristics of a selection of TLM frameworks for SystemC. For an overview on the characteristics of the communication architectures which are 'abstracted' by these, see appendix A.

OSCI TLM

Founded in 2003, the OSCI TLM Working Group (TLM-WG) pioneered the development of TLM frameworks for SystemC with the release of the OSCI TLM kit 1.0 [109]. A set of three interfaces is provided that form the heart of the kit, enabling unidirectional blocking, unidirectional non-blocking, and bidirectional blocking transfers.

The TLM 1.0 kit does not define standard data types nor abstraction levels. The intended use of the kit is to develop customized channels with it, using application-specific data structures and user-defined protocols. Thus, the OSCI kit does not help in achieving model interoperability.

Having identified these issues, the OSCI TLM Working Group is currently working on a revised version of the kit (OSCI TLM 2.0). A first proposal of concepts has been made available for public review in spring 2007 [90]. It involves several of the proposals that I present in this PhD thesis, as the GreenSocs initiative (which I am an active member of) has submitted GREENBUS to OSCI.

OCP SystemC channels

The OCP-IP provides a comprehensive library of point-to-point channels for modeling of SoCs based on the Open Core Protocol with SystemC. Three levels of abstraction are supported, namely OCP-tl0 for cycle accurate, OCP-tl1 and OCP-tl2 for CCATB, and OCP-tl3 for PV communication modeling.

A large set of interface methods for both blocking and nonblocking channel access is provided. OCP channels use predefined data types for transfer qualifiers such as address, command, thread identifier, etc. and provide basic instrumentation for transaction monitoring. However, they do not provide any means for communication architecture simulation, as they only support point-to-point communication.

ST Microelectronics TAC

TAC, 'transaction accurate communication', is a set of channels and interfaces aiming at the creation of virtual PV prototypes for early software development. It is built on top of OSCI TLM 1.0 and provides a pair of so-called initiator and target ports. These ports implement simple blocking `read` and `write` communication methods.

As part of TAC a router model is delivered that can add `wait` calls to transactions so that approximate (PV+T) communication times are estimated, e.g. using weighted randomization. However, TAC does not support communication architecture simulation.

SystemC^{SV}

An interesting approach to TLM communication modeling at mixed levels of abstraction is the SystemC^{SV} extension of SystemC [116]. It defines an interface that describes communication among PEs at different levels of abstraction in terms of *interface items*. An interface item may represent

either a complete transaction, a frame or word, a field within this frame, or a signal state on a physical wire.

SystemC^{SV} provides C++ macros with which the composition of interface items can be described in a declarative style. Interfaces use these macros to automatically decompose interface items for transmission and reassemble them on reception. Thus, mixed multi-level communication of PEs at different abstraction levels is made possible.

While the SystemC^{SV} approach is very appealing for describing point-to-point communication, it does not support modeling of shared-buses. Also, the simulation speed-up of 160x ([116]) when using abstract interface items instead of RTL level interface items is considerably lower than the performance reached by other TLM frameworks (including the results of this thesis).

OCCN

OCCN, ‘on-chip communication network’, see [25]) addresses network-on-chip modeling with SystemC at mixed abstraction levels. It transports ‘protocol data units’ (PDUs) among PEs. PDUs are composed of user-definable fields that carry user data and protocol information. An OCCN channel implements blocking send and receive methods for PDUs. Receive delays and timeouts can be specified as parameters, enabling PV+T modeling equal to ST TAC.

Coppola et al. propose the implementation of an application API on top of the PDU transport layer. This API can provide master and slave ports with convenience methods. Similar to OCP channels, the OCCN channel is a point-to-point channel that does not support communication architecture simulation.

OSSS library

The goal of the OSSS library from OFFIS Oldenburg [42] is to enable HW/SW communication modeling with synthesizable channels. OSSS supports a two-layered channel model. Internally, a set of `sc_signals` is used, which is equivalent to RTL wires and therefore synthesizable. An application layer provides more abstract convenience methods.

For bus modeling a predefined `read/write` interface is proposed. A protocol library contains different implementations of these interface methods, e.g. to simulate the OPB. Data transport and arbitration in OSSS is performed at the RTL level of abstraction. Thus, bus simulation is cycle accurate. The simulation performance achieved with this approach hence is hardly better than with pure RTL models. HW/SW communication synthesis is not discussed in [42].

IBM CoreConnect models

IBM provides SystemC bus functional models for their CoreConnect architecture. They enable precise CC simulation of the PLB [56], OPB [55], and DCR [54] buses. For each of the three buses a dedicated API is provided, which are quite comprehensive and low-level: for example, the PLB API boasts 33 different interface methods. Thus, the IBM library is a good tool to create a virtual prototype of an already existing CoreConnect SoC, but is inappropriate for architecture exploration.

CoWare AMBA models

A library of AMBA [5] bus functional models is provided with CoWare’s Platform Studio, which is a commercial SystemC design environment. These models enable simulation of AHB [8] and AXI [7] on-chip buses, but similar to IBM’s models their dedicated AMBA-specific APIs hamper architecture exploration.

CCATB AMBA models

In their paper on the CCATB simulation approach [103], Pasricha et al. present CCATB CAFMs for the AMBA AHB and AXI buses and compare them with ARM’s cycle accurate bus functional models. A speedup of 55% is achieved, which is rather unsatisfying in comparison to the other approaches. This limited result may be due to the utilized instruction set simulator.

ROM

Result oriented modeling (ROM [113]) too takes advantage of the limited observability within a transaction to increase the performance. The idea is to calculate the transmission time of a transaction a priori and then advance simulation time to the end time of the transaction. This is similar to PV+T approaches such as TAC, but in addition, ROM checks whether a ‘disturbing influence’ has occurred, such as an overlapping transaction on a shared bus. In this case, corrective measures are taken if necessary.

Schirner et al. achieve simulation speeds close to that of PV models with this approach. However, ROM simulation results are only correct when observed at transaction boundaries. Thus, BA and CC models are not supported. Moreover, ROM CAFMs are only periodically synchronous to SystemC simulation time.

Kogel et al.

In [78], Kogel et al. present a a combined bus/NoC simulation framework for packet based communication via a so-called NoC channel. Multiple masters and slaves can be connected to the channel. It implements a symmetric request-response communication scheme. Masters initially not compliant to this interface can be attached by means of adapters.

An advantage of Kogel’s approach over all of the above proposals is that protocol simulation and multiple access arbitration is inherently supported by the channel. So-called network engines can be attached to the channel that implement the behavior of a bus or NoC protocol. Upon initiation of a transaction, the channel creates a transaction data structure and forwards it to the network engine. The network engine collects concurrent transaction requests in a queue and delays them in accordance with the arbitration policy. Transmission times are taken into account as well, however on a coarse-grained packet basis only. The framework therefore lacks the possibility of providing CCATB timing estimations. Communication refinement towards the implementation model is not considered by the authors.

3.3.1 Summary

Table 3.1 summarizes the characteristics of the reviewed TLM frameworks.

3.4 Discussion

TLM frameworks remain one of the most required and most argued over tools in system-level design. There are a number of different technological features of different proposals. They differ either in the supported levels of abstraction or the supported transportation technology. There is even argument about the requirement and scope for a ubiquitous framework. We believe that, given the number of proprietary CAFMs ([9, 10, 19, 26, 41, 57, 103, 112, 132]) and unpublished ‘generic buses’³ present in the industry today, the utility is unquestionable. From my research cooperations with GreenSocs, IBM, Intel, Texas Instruments, and Volkswagen it became clear that significant productivity gains in TLM-based SoC design can be achieved if there would be industry-wide agreement on the fundamental techniques of TLM frameworks.

The survey shows that there are currently two main approaches to such frameworks:

1. ‘generic’ frameworks such as OSCI TLM and ST TAC enable flexible high-level communication modeling but do not support simulation of architectural details;
2. CAFM libraries such as IBM’s CoreConnect models for SystemC enable accurate simulation of a particular communication architecture.

³Due to feedback I got from discussions in the GreenSocs community, the OCP-IP SLD Working Group, and the OSCI TLM Working Group, in the industry there currently are a lot of proprietary ‘niche’ solutions in use.

	Simulation accuracy		Access semantic	Transfer types		Data types	Pass-by-...	Supported interconnects	
	Transactions	Protocol phases							
CCATB AMBA models									
IBM CoreConnect TLM		✓	blocking	word	burst	fixed	pointer	p2p	bus NoC
		✓	non blocking	✓	✓	✓	value	✓	✓
Kogel et al.	✓		✓ (slave)	packet		✓	pointer	✓	✓
OCCN	✓		✓	packet		✓	value	✓	✓
OCP TL 1		✓	✓	✓	✓	✓	value	✓	✓
OCP TL 2		✓	✓	✓	✓	✓	pointer	✓	✓
OCP TL 3	✓		✓	✓	✓	✓	value	✓	✓
OSCI TLM	✓		✓	✓	✓	✓	value	✓	✓
OSSS Channels		✓	✓	✓	✓	✓	value	✓	✓
SystemC-SV	✓		✓	✓	✓	✓	value	✓	✓
TAC	✓		✓	✓	✓	✓	value	✓	✓
ROM	✓			✓	✓	✓	pointer	✓	✓

	Available models		Configuration		Analysis features
			API	File	
CCATB AMBA models	AXI, AHB				
IBM CoreConnect TLM	PLB, OPB, DCR		✓		Proprietary transaction tracing
Kogel et al. Simple bus examples					
OCCN STBus					Packet recording (callback functions)
OCP TL 1	P2P channel		✓	✓	VCD trace, performance/trace monitor
OCP TL 2	P2P channel		✓	✓	Performance/trace monitor
OCP TL 3	P2P channel		✓	✓	Performance Monitor
OSCI TLM	FIFO, Simple bus example				SCV
OSSS Channels	OPB		✓		VCD Traces
SystemC-SV	CAN, USB				VCD Traces
TAC	STBus				SCV and message trace
ROM	AXI, AHB, CAN				

Table 3.1: Characteristics of reviewed TLM frameworks

Almost all frameworks I have reviewed are either in the first or in the second category. Only few approaches support both design tasks at once. One reason is that many frameworks only deal with one specific level of abstraction.

TLM engineers face a design flow gap between the golden model and the refined TLM models (virtual prototypes): model-specific adapters need to be in place that translate between the high-level and the CAFM interfaces, for example if a TAC PE is to be connected to IBM's CoreConnect CAFM. Literature review and web scans evince that such adapters are not available at large. Thus, there is no way to avoid manual adapter development, which is tedious and error prone.

3.4.1 Towards a TLM framework interoperability standard

We believe that significant productivity gains in TLM-based SoC design can be achieved if there would be industry-wide agreement on the fundamental techniques of TLM frameworks. The most important result of my reviews of the existing frameworks is that the part of the TLM framework implementations that in fact needs to be standardized is surprisingly small. The experiments in the following chapters will show that IP cores at different levels of abstraction and with different interfaces can work together over various channels and CAFMs if the following technical fundamentals are carefully considered:

- Data representation;
- Transport mechanism.

The outcome of this research is the GREENBUS TLM fabric for SystemC, which combines a standardized yet extensible data representation with a generic transport mechanism for modeling protocol phases. The scope of this work is to enable both mixed mode (different levels of abstraction) and heterogeneous (different bus interfaces) modeling, using the channels and CAFMs that meet best the respective requirements of the system engineer. Hence, one design goal for GREENBUS is to provide better user-defined adaptability than other frameworks. The most interesting work has been done by Kogel et al. Their generic interconnect model for simulating on-chip busses and networks-on-chip enables architectural exploration. However, they deal with this at a very high level of abstraction only and therefore do not support cycle-count accurate timing estimations. The CCATB models presented by Pasricha et al., on the other hand, provide satisfying timing fidelity, but do not offer the simulation speed required for early software development and fast architecture exploration. Moreover, they are available for the AMBA buses only. Other research groups identified the need of generic bus models but do not offer solutions [22, 39].

The reviews of CAFMs being used in the industry show that at higher levels of abstraction (PV) there is industry wide cohesion, with blocking calls being used. OSCI and ST suggest the usage of a single transport call for PV. At lower levels of abstraction nonblocking interfaces are often preferred, but not ubiquitous. For example frameworks such as the OCP channel library offer both. The fundamental requirement is to provide a mechanism to efficiently transmit data, and timing information from master to slave.

The first aspect of this is how memory will be managed. The choices are to either have the master port allocate enough memory for both the request, and response information (from the target) – this is commonly called *pass-by-pointer* as recommended by CoWare and ST Microelectronics [41, 119]; or, all data can be transferred as data items and no memory allocation is necessary – this is often referred to as *pass-by-value* [88, 109].

This fundamental difference at the outset of a transaction impacts the way subsequent communication is handled. TLM fabrics that deploy *pass-by-pointer* can then either use subsequent method calls or simply events to indicate updates to the (shared) data structure. Fabrics that pass by value must use method calls to pass updated values.

The second aspect is the timing of data transfers. In order to minimize the amount of re-calculation, some bus fabrics are designed to execute on the falling edge of the clock ([42, 61]), such that all requests

which need to be arbitrated will be present, and the arbitration only need take place once. Compelling as this scheme at first seems, on today's multi-bus SoC's, designers soon run out of 'falling edges', and interfacing to RTL becomes very much harder. Another approach is to recalculate results when a 'timing violation' is detected ([113]), which achieves high simulation speed but does not work with CCATB and RTL models.

One of the goals of this work is to be able to support the body of existing IP, which uses the full spectrum of 'bus interfaces', hence I introduce an extra requirement on GREENBUS to be able to support both blocking and nonblocking interfaces, pass-by-pointer and pass-by-value. For blocking interfaces, GREENBUS should support a PV mode of operation. For nonblocking interfaces, it should enable both CCATB and more accurate simulation of different communication architectures.

Similarly, there are different approaches taken to the data that is transported. There are in essence two different approaches taken to this subject. Some favour extensible data types, while others opt for providing a defined bus fabric onto which 'all other' buses can be mapped. The latter approach is typical of bus vendors, while the former is often adopted by bus users. Hence, ST's TAC favors extensible data types, enabling the user to transfer any data, while for example OCP predefines the data structure (bit vectors) but offer some user definable flags.

The disadvantage of the extensible data structures is, first there can be lack of consistency between implementations, second many data types do not lend themselves to extension. However, there are equal disadvantages with defined busses, i.e. its extension if (and when) a bus is being used which does not easily map onto the offered structures. In addition, from the simulation speed perspective, a pre defined 'common bus' incurs a simulation overhead if modes have to be wrapped onto the predefined structures.

The approach presented in this thesis is to fix the data structure elements, but to allow choice of which elements to use in a channel. The intent is to mitigate the problems of inconsistent implementations while offering the user a flexible channel and CAFM development framework.

Both ARM's AMBA and IBM's CoreConnect offer interfaces and protocols that can deal with most communications occurring in embedded systems. Both of these protocols map easily onto the OCP bus fabric. The details of how they map onto GREENBUS will be described in the next chapter.

4 The GreenBus Approach

Contents

4.1	Introduction	33
4.2	Requirements	34
4.3	General concepts	35
4.4	Bus accurate abstraction approach	37
4.5	A data representation and transport mechanism for TAQ	43
4.6	Connection layer	48
4.7	Generic protocol	56
4.8	GreenBus extensions and interoperability	62
4.9	Adaptation layer	69
4.10	Experiments	71
4.11	Summary and outlook	77

4.1 Introduction

In the last chapter, three key issues have been identified that need to be standardized to enable systematic and efficient communication architecture design with SystemC:

1. Communication abstraction;
2. Data representation;
3. Transport mechanism.

The goal of this chapter is to develop an appropriate answer to the question how such a standard should look like. The results have been implemented in the GREENBUS TLM fabric for SystemC. There are three basic parts to GREENBUS fabric:

1. A *generic protocol* definition, to be used as a base for abstract channel *and* phase-based CAFM development. This includes both the representation of the data that is transported across the channel and the modeling of protocol phases and their synchronization with the SystemC simulation time.
2. An *adaptation layer* which enables users to define their own bus independent API. This provides the possibility to connect heterogeneous PEs to a GREENBUS CAFM, independent of their interfaces.
3. A *communication router* which is bus and user API independent. The router takes the application-specific protocol definition and provides a well implemented fast communication router which works at several abstraction levels.

The advantage of this approach is that it enables IP reuse, as the user API is independent of the bus fabric, and as standards are developed (i.e. the OSCI TLM standard which is currently being worked on [90]) GREENBUS can be adapted to remain compliant with it, while user code can remain unchanged.

4.2 Requirements

This chapter develops a list of main requirements for GREENBUS. Starting with

- R1 Standardization proposal.** Develop methodologies and proof-of-concept how a generic TLM fabric for SystemC should look like.

we define the following two additional goals:

- R2 Ease of use.** The API and components of GREENBUS shall be self-explanatory and fault safe, and it shall be easy to make use of and extend GREENBUS functional abilities.
- R3 Performance.** The simulation performance when using GREENBUS for communication architecture exploration shall not be considerably less than with dedicated CAFMs.

Table 4.1 works out the details of R2. This table forms the wish list of user-view features which this PhD research is dedicated to, aiming at as efficient embedded system design with TLM and SystemC as possible. Initially, these requirements were derived from reviews of the related work. During the developing process of GREENBUS, I refined some of them in respect of experimental results and due to feedback I got from discussions with TLM engineers in the chip industry.

The performance demand R3 can be understood in two different aspects: first, in each framework there is a trade-off between performance and usability/flexibility, which needs to be carefully considered; second, to be successful it is extremely important for a standardization proposal that it does not come with the price of poor performance.

Table 4.1: Detailed user-view requirements for GREENBUS

Req.	Description
R2.1	Support different user APIs. This is important to attain interoperability of heterogeneous IP. Any PE shall be attachable to a GREENBUS interconnect independent of its interface and protocol.
R2.2	Generic protocol for data access and transport. The low-level transport and data structures shall be separated from the user API layer by an intermediate generic protocol layer. Thus, the bus core protocol can be replaced while user code remains unchanged.
R2.3	Mixed-mode modeling, avoid adapters. GREENBUS shall make it easy to mix PEs at different abstractions in one model. Writing model-specific communication adapters shall be rendered unnecessary.
R2.4	Enable both untimed performance and timed architecture simulation with one model. Separate blocking and nonblocking transport interfaces shall be provided such that a PE can comprise two distinct implementations, one for fast untimed and one for accurate timed simulation.
R2.5	Automatic memory management. All transaction related data structures shall be automatically allocated and deallocated by the fabric.
R2.6	Clearness about transaction data validity and life span. There shall be no doubt when transaction data is valid and when it is safe to read/write specific data elements.
R2.7	Clearness about abstraction. The API shall support different abstraction models for communication, which shall be clearly distinguishable by the user and thus encourage systematic ‘abstraction modeling’.
R2.8	Clearness about communication timing. The API shall support an appropriate method for specification of communication delays at different transaction time points (depending on abstraction) and at any location on a transaction’s path through the system.
R2.9	Provide a simple way of constructing interconnects. The user shall be able to connect PEs via point-to-point links, buses, and Networks-on-Chips with a simple script-like description syntax both in the testbench (compile time) as well as using a platform configuration file.
R2.10	‘Plug-In’ interface for different CAFMs. The user shall be able to replace the communication architecture functional model without the need for re-compilation of the system model.
R2.11	Facilitate development of new user APIs and CAFMs. The fabric shall define a methodology that enables straight-forward implementation of user extensions and additional simulation abilities.
R2.12	Analysis & debug. Support for communication analysis and debug shall be inherent to the GREENBUS fabric. Thereby adhere to common standards such as SCV [99].
R2.13	Interfacing to external tools. There shall be a flexible mechanism for connecting external development tools to the fabric, such as CoWare Platform Architect [26] and ModelSim [87].

4.3 General concepts

GREENBUS is an open-source TLM fabric built on top of the SystemC-2.1 library [97]. In the following we use the word ‘GREENBUS’ both as the name for this TLM fabric and as short term for a TLM interconnect that has been constructed with it (e.g., ‘*both JPEG encoder and DDR RAM are connected to a PLB GREENBUS*’ means that the two PEs ‘JPEG encoder’ and ‘DDR RAM’ are connected to a PLB CAFM that has been constructed with the GREENBUS TLM fabric). GREENBUS supports both point-to-point communication modeling with abstract channels and communication architecture exploration with CAFMs. Point-to-point channels can be easily replaced by CAFMs, and CAFMs can be configured to simulate different bus architectures. Any of these model modifications do not necessitate changes to the PEs nor do they require the designer to re-compile his model. Different communication architectures can be explored by simply making changes to a configuration file.

The PEs themselves use ‘convenience methods’ for exercising transactions. As has been seen in chapter 3.2, the convenience interface that is chosen varies between different user companies, and different CAFMs. Thus, GREENBUS follows a layered approach that decouples the PE communication interfaces from the GREENBUS interface – which is the underlying low-level transport API, denoted *connection layer*.

This has several advantages:

- PE models can be written using the most convenient IO interface API, and can remain unchanged (even shipped as binary objects).
- PEs can be exchanged by simply changing the ‘user API to GREENBUS connection layer’, which we refer to as *adaptation layer*.
- The adaptation layer is efficiently managed and has no significant effect on simulation performance.

Figure 4.1 shows a simple use case of GREENBUS to illustrate this approach. Four communication layers can be distinguished, with the two middle ones belonging to GREENBUS (table 4.2). The **master port** on the left-hand side provides an application-specific communication interface to the master PE, referred to as the master **user API**, here the **read/write** methods of ST TAC. Slave

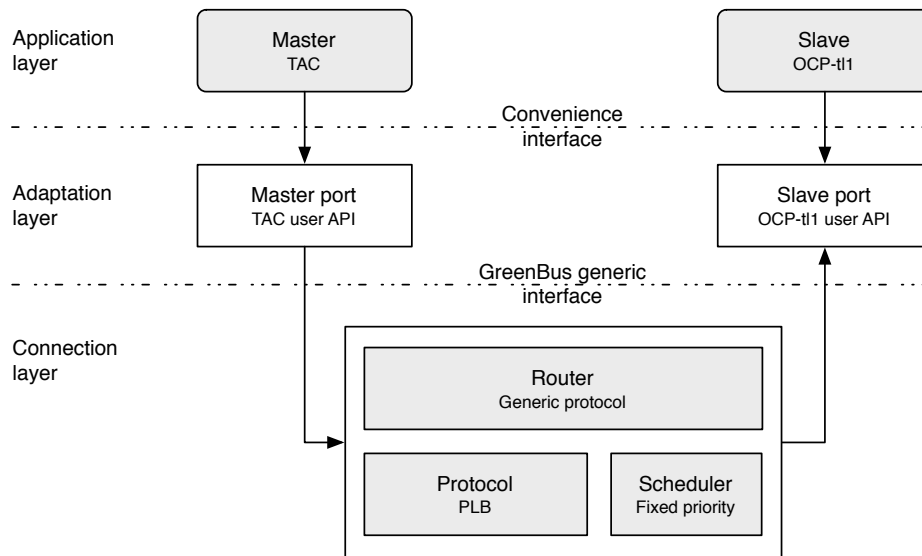


Figure 4.1: Layered communication approach

Table 4.2: Communication layers in GREENBUS-based TLM models

Layer	Description
Application	Processing elements with dedicated interfaces and protocols.
Adaptation	Master and slave ports map PE interface methods and protocols to the generic GREENBUS interface and protocol.
Connection	Generic low-level data structures and transport mechanism, bus functional simulation.
SystemC	Basic data types, events, ports, ...

PEs are connected to the channel by the same mechanism. A slave implements a potentially different application-specific interface defined in the slave user API, which is implemented in the *slave port*. The underlying interconnect is independent from the user APIs. User APIs can be chosen by the model designer and can be different for each PE in the system. Several typical user APIs have been implemented for GREENBUS. For example, the OCP slave user API in figure 4.1 supports the same interface methods than the SystemC channels provided by OCP-IP [95]. Hence, PEs that have been designed for the Open Core Protocol can be connected to a GREENBUS interconnect with this user API.

For CAFM development, a router component is used, which is part of the GREENBUS fabric¹. It is accompanied by a bus simulation engine, which is responsible for all the bus protocol arbitration and timing estimation. The bus simulation engine comprises user-exchangeable protocol and scheduler classes. For example, in the system presented in figure 4.1 an IBM Processor Local Bus is simulated.

The router and bus simulation components can be left out to create a simple point-to-point channel. Then master and slave port are directly plugged together, which is the simplest use model of GREENBUS. An example system that employs both use models of GREENBUS – point-to-point channels and CAFMs – is shown in figure 4.2.

¹The GREENBUS core elements are described in this chapter, the router is described in chapter 5.

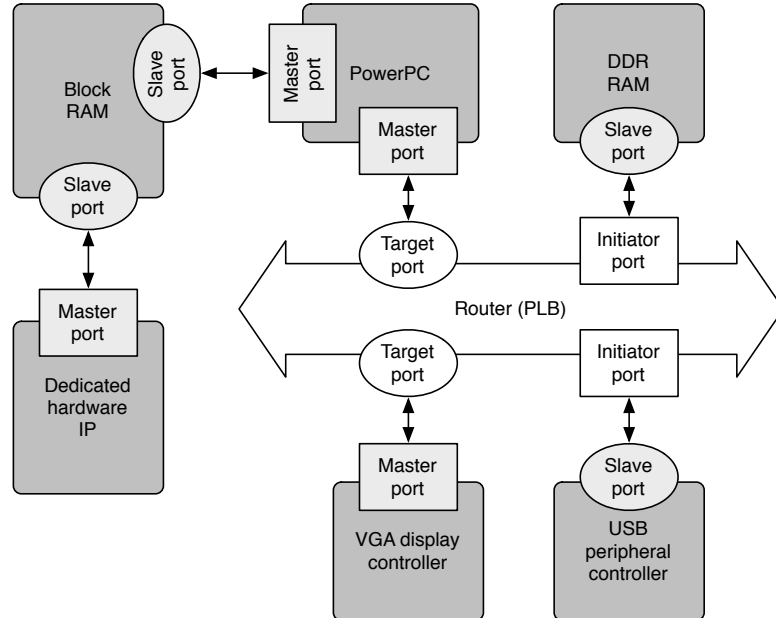


Figure 4.2: Example system model with both point-to-point channels and a CAFM based on GREENBUS

Note that four different port types are distinguished:

- Master ports and slave ports implement the individual user APIs for the PEs. Each PE in the example may use another user API.
- **Initiator ports** and **target ports** provide access to router components, which implement the generic interface and enable communication architecture functional simulation.

Master and slave ports are possessed by the PEs. They are instantiated in the PE constructors and equip them with a generic interface to the outside world. Inside the PEs, however, the user API(s) implemented by their master/slave port(s) is used for communication.

The connections between master and slave ports are set up in a configuration file. There, also CAFMs can be instantiated. When a master or slave port of a PE is connected to a CAFM, an appropriate target or initiator port is automatically created in its router. Listing 4.1 shows the configuration file for figure 4.2.

Listing 4.1: Configuration file to set up the interconnect in fig. 4.2

```

1  BUS(plb, PLB, 10, SC_NS)
   CONNECT(PowerPC.plb_port, PLB.target_port)
3  CONNECT(PLB.init_port, BlockRAM.port2)
   CONNECT(PowerPC.bram_port, BlockRAM.port1)
5  CONNECT(myHW.bram_port, BlockRAM.port2)

```

4.4 Bus accurate abstraction approach

As has been pointed out in chapter 3, finding an appropriate approach to communication abstraction is crucial to enable fast yet accurate communication architecture simulation. The here presented approach is based on the assumption that transactions over an RTL communication architecture model can be simulated as a sequence of protocol phases in order to get a more abstract yet accurate enough CAFM (see 3.2.2). In the following, a protocol phase data representation and transport mechanism for SystemC is developed with which such CAFMs for various protocols can be modeled.

4.4.1 Transactions, atoms, and quarks

We refer to the implementation approach used in GREENBUS with the term **transactions, atoms, quarks** (TAQ). Transactions are modeled as a sequence of so-called atoms. They are a SystemC representation of protocol phases. An atom has two time points, atom start and atom end, and refers to ('carries') a number of so-called quarks. They are a SystemC representation of the information content of protocol phases. In a CAFM based on atoms and quarks, an atom is the smallest un-interruptable part of a transaction that once started will complete its life cycle.

4.4.1.1 Atoms

All the bus and IO structures we have analyzed², namely OCP [94], AMBA [5–8], CoreConnect [54–56], Wishbone™ [49], STBus™ [25], PCI Express [105], CAN™ [108], and USB [127], can be represented as containing transactions with up to just three different types of atoms: *init* atoms, *data* atoms and *finalize* atoms.

- The init atom carries all the transfer qualifiers, after its completion both master and slave are ready to exchange data.

²See appendix A for an overview of the analyzed communication architectures.

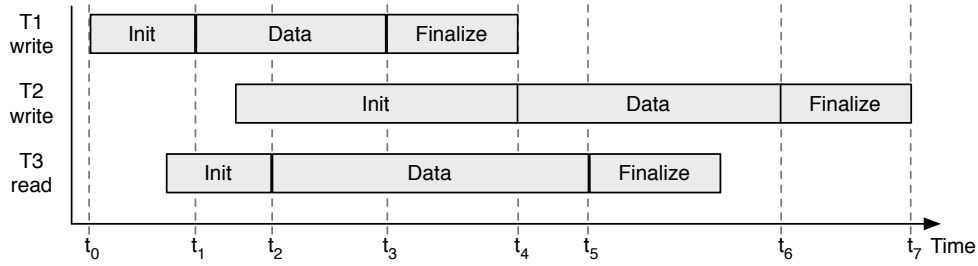


Figure 4.3: Overlapping atoms of concurrent PLB transactions

- Data atoms are used to transfer write or read data and all accompanying qualifiers like byte enables or error flags.
- The finalize atom finishes the transfer and can carry final responses or information needed to release the transaction properly.

It is possible that a transaction does not use all atoms, e.g. there are buses that won't need a finalize atom, a simple IO interface may only use one init atom. It may furthermore be the case that several init, data, or finalize atoms are used.

Atoms may overlap in time. Consider figure 4.3. It shows the atoms of three transactions on a CAFM of the IBM Processor Local Bus (PLB). The first transaction, T1, is a SRMD write and starts at time point t_0 . T2 is another SRMD write, issued by a second master while T1 is in the data phase. Hence, an init atom starts in parallel to the data atom of T1. However, since the PLB can only handle one write transaction at a time, it is dynamically delayed by the CAFM until the finalize atom of T1 has been terminated, indicating that the write bus has been deallocated. The third transaction, T3, is a read transaction and can run in parallel to the write transactions, as the PLB provides separate read and write buses. The dashed lines in the graph show the time points that need to be considered by the CAFM in order to simulate the timing of these communications with CCATB accuracy. Note that the number of time points simulated is independent from the number of clock cycles covered by the simulation. It only depends on the number of atoms.

4.4.1.2 Quarks

A quark is nothing more than a basic data type. A fundamental principle of GREENBUS is that quarks are pre-defined. We simply suggest that for all features of a communication protocol there should be a one-to-one mapping of feature and underlying transport type. For instance, any bus CAFM capable of transporting byte-enable information with the data should always use the same quark to do so. This fundamental principle is the key to providing model inter-working with the minimum cost.

The quark data types need not be exhaustive, as bus and IO features which are really unique will always require some interpretation between IP not designed to the same interface. In this case, inter-working will always come with some cost, hence standardizing on the types for unique features does not help.

We can distinguish different categories of quarks required for communication architecture simulation:

- *Protocol quarks*: they carry control flow information of the modeled protocol;
- *Payload quarks*: they model the data flow among the communicating PEs;
- *Simulation quarks*: they provide meta information which cannot be found in the RTL model but is required for its abstract simulation, e.g. for communication analysis.

Table 4.3: Standard quark set

Quark	Description	Type
Init quarks		
MAddr	Target address	uint64
MAddrSpace	Master address space	uint64
MAtomicLength	Length of atomic burst	uint32
MBurstLength	Number of data words in this burst	uint32
MBurstPrecise	Given burst length is precise	bool
MBurstSeq	Address sequence of burst	GenericMBurstSeqType
MBurstSingleReq	Burst uses single request / multiple data protocol	bool
MCmd	Master command (read, write, ...)	GenericMCmdType
MConnID	Master connection identifier	uint32
MID	Master identifier	uint64
MReqLast	Last request in a multiple request burst	bool
MTagID	Master tag identifier	uint32
MTagInOrder	Force tag-in-order	bool
MThreadID	Master thread identifier	uint32
MReqInfo	Extra information sent with the request	uint64
TransID	Unique transaction ID	uint64
Data quarks		
MSData	Transaction data	GBDataType
MDataInfo	Extra information sent with the write data	uint64
SDataInfo	Extra information sent with the read data	uint64
MSByteEn	Byte enable information sent with the burst	uint32
MSBytesValid	Number of bytes valid in MSData	Data
SResp	Slave response to master command	GenericSRespType
SThreadID	Slave thread identifier	uint32
STagID	Slave tag identifier	uint32
STagInOrder	Force tag-in-order	bool
SRespInfo	Extra information sent with the response	uint64
SRespLast	Last response in a multiple request burst	bool
Finalize quarks		
Error	Error identifier	uint32

Table 4.4: Mapping of PLB and AXI onto GREENBUS quarks and atoms

Quark	Atom	AXI	PLB
Init atom			
MAddr		AWADDR / ARADDR	PLBABus
MID		-	PLBMID
MTagID		AWID / ARID	-
MBurstLength		AWLEN / ARLEN	Used with line reads/writes and fixed length bursts
MBurstType		AWBURST / ARBURST	PLBSize, PLBrdBurst / PLBwrBurst
MByteEn		AWSIZE / ARSIZE	PLBBE
MCmd		Fixed parameter of port	PLBrNw
userQuark1		AWLOCK / ARLOCK	PLBbusLock
userQuark2		AWCACHE / ARCACHE	PLBtype
userQuark3		AWPROT / ARPROT	PLBordered, PLBblockErr PLBguarded, PLBcompress
	✓	AWVALID / ARVALID	PLBrequest
	✓	AWREADY / ARREADY	PLBAddrAck
userQuark4		N/A	PLBreqPri
Data atom			
MID		-	PLBMID
MTagID		WID	-
STagID		RID	-
MData		WDATA / RDATA	PLBrdDBus / PLBwrDBus
MByteEn		WSTRB	PLBBE
MSBytesValid		WLAST / RLAST	Inverted PLBwrBurst / PLBrdBurst
	✓	WVALID / RVALID	Write Data is always valid during burst / Read Data is valid with PLBrdAck
Finalize atom			
MID		-	PLBMID
MTagID		WID	-
STagID		RID	-
	✓	BRESP / RRESP	-
	✓	BVALID / RVALID	-
	✓	BREADY / RREADY / WREADY	PLBwrComp / PLBrdComp
	✓	-	Combination of PLBBusy, PLBErr

As an initial set, we believe that the set of types defined by OCP [94] is relatively comprehensive, with some minor additions. Table 4.3 lists the quarks defined with GREENBUS. The quarks are loosely organized in three groups to show their typical relation to the corresponding protocol phases and therefore to the corresponding atoms. However, this relationship is not mandatory. In several cases it may make sense to relate an init quark to a data or finalize atom, whereas in other cases data quarks may be used along with init atoms as well (e.g. a CAFM may allow to transfer the first data word as part of a write transaction init atom).

It is important to note that the data types shown in the ‘Type’ column in table 4.3 are not fixed. They are a proposal that I have used in my GREENBUS reference implementation. To achieve interoperability, however, not the data types but set and get methods for reading and writing these data types should be standardized. As long as the syntax and the semantics of these methods do not change, the underlying data types may differ.

In general, each signal in an RTL model can be represented by a corresponding quark. But in many cases it also makes sense to subsume several RTL signals in a single quark, to get a more abstract data representation. Furthermore it is important to understand that quarks are only validated at the atom start and end time points, but not in-between. Hence, in a TAQ model value changes of an RTL signal *during* the life cycle of an atom need to be considered spatially, not temporally. That is, if a data burst is modeled with a single atom, it must carry so many data quarks as there are individual data transfers in the RTL model during the burst. Again, the individual data word quarks also can be subsumed in a single quark that is a vector of data words.

The quarks in table 4.3 allow for modeling the basic communication features, such as single-beat reads/writes and burst transfers of fixed as well as variable length. To illustrate this approach with an example, table 4.4 shows the quarks and atoms needed for IBM’s CoreConnect PLB and ARM’s AMBA AXI. Their common transfer qualifiers are mapped onto standard quarks, uncommon features are mapped onto so called user quarks, whose semantics are defined by the users. While for PLB, four user quarks are required, AXI needs three. Inter-operation between PLB and AXI PEs is enabled with the standard quarks, which are sufficient for basic communication. Transactions between PLB-only or AXI-only PEs, however, can make use of the additional user quarks to enable special protocol features, such as bus locking or dynamic priority. Chapter 4.8 will give a deeper discussion of this extension mechanism.

The column entitled ‘Atom’ in table 4.4 shows a check mark for those bus signals that are already implicitly modeled by the atom type and therefore do not require an explicit quark. For example, the WVALID signal of AXI indicates that the master write data is valid. In a GREENBUS CAFM, this information can be modeled by starting a data atom and relating it to the corresponding MData and further quarks. To model the WVALID signal value change no extra quark is needed, as this information is already indicated by the start of the atom. Similarly, the acknowledge signals shown in the last section of the table can be modeled by starting a finalize atom.

4.4.2 Abstraction level formalism

A natural outcome of considering transactions as being composed of atoms and quarks is that we can present a formalism for TLM communication abstraction.

Explanation 8. Depending on the time and data abstraction a GREENBUS channel uses to execute communication, it implements one of the following levels of abstraction:

- *Programmers view (PV)*: a PV model is communication with atomic transactions, i.e. each transaction is a single atom which carries all quarks encompassed by the transaction. Exactly two time points are modeled, transaction start and end. There are no intermediate time points considered by a PV model, especially quark value changes are only evaluated at transaction start and end.

- **Bus accurate (BA):** with the term bus accurate I refer to models that are communication with several atoms. That is, in a BA model transactions are composed of a sequence of atoms and each atom carries a subset of the quarks encompassed by the transaction. For each atom, its start and end time point are modeled, and at each of these time points, quark value changes may be evaluated.
- **Register transfer level (RTL):** an RTL model is communication with atoms and clocked quarks, i.e. in addition to transactions being composed of a sequence of atoms, quark value changes are not only evaluated at atom start and end time points, but also at each cycle of a reference clock signal.

Table 4.5: Abstraction / time resolution relationship

Abstraction level	Time points of quark validations
PV	Transaction start and end
BA	Atom starts and ends
RTL	Atom starts and ends, and quarks at each clock edge

Table 4.5 lists the time points of quark validations at the different levels of communication abstraction. In accordance with this, in a GREENBUS simulation a PV master will always send the whole transaction including all quarks at once and wait for it to be completed. A BA master would send the transaction atom-wise, to keep control of protocol phase propagation and a RTL master would also send the transaction atom-wise but in addition keep control of the quark values at each cycle of a clock signal.

4.4.2.1 Simulation accuracy and terminology

With the implementation technique for TAQ presented in the following, SystemC channels and in particular CAFMS can be developed using the above levels of abstractions. But the mere fact that a CAFM is a PV, BA, or RTL model does not provide any information about its simulation accuracy. Hence, we in addition need to examine a BA model regarding the following interesting characteristics:

- A BA model can have the property to simulate the underlying RTL model with *cycle count accurate at transaction boundaries* (CCATB) timing accuracy (see 3.2.2). In terms of the SystemC simulator this means that when a transaction has been started, the model will notify the end of the transaction at the correct simulation time in the future, with reference to a given communication clock speed. We will see that for CCATB accuracy typically only two atoms are required.
- A CCATB accurate BA model in addition can have the property of notifying the starts and ends of the simulated protocol phases at the correct simulation times. We refer to this timing accuracy with the term **cycle count** (CC) accuracy. The experiments below will show that for the examined buses CC accuracy is achieved by splitting the data atom into a sequence of data atoms, each modeling the transfer of one data word (its size depends on the data bus width).
- Finally, a CC accurate BA model can have the property to be cycle accurate, thereby providing the same timing fidelity as the simulated RTL model. To this end, in addition to atom notifications quark value changes must be notified with each cycle of a clock signal. This approach has been examined in the first GREENBUS version but was cancelled due to lack of performance.

Figure 4.4 illustrates how transactions can be built out of atoms and quarks and shows the points of interest (depicted as arrows) at the various abstraction levels. To simplify the terminology for describing the abstraction level and timing accuracy used by a CAFM to simulate transactions, in this figure and in the following we use the following nomenclature:

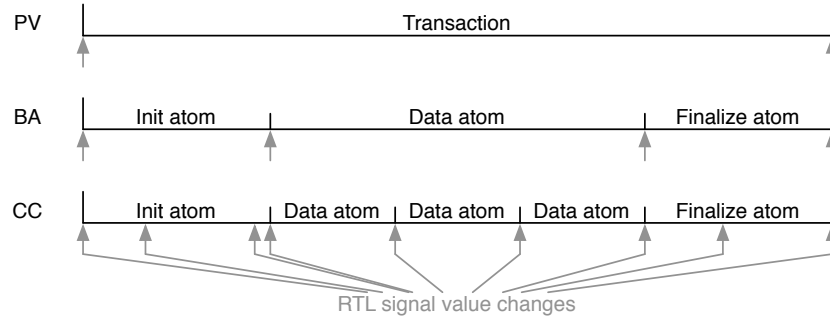


Figure 4.4: Simulation of RTL transactions with atoms and quarks

- *PV transaction*: the transaction is simulated by a PV model with PV ‘accuracy’.
- *BA transaction*: the transaction is simulated by a BA model with CCATB accuracy.
- *CC transaction*: the transaction is simulated by a BA model with CC accuracy.

4.4.2.2 Atom life cycle

Since atoms in GREENBUS are the basic blocks to build CAFMs with, it is important to understand their life cycle in the simulation. The life of an atom starts with it requesting access to the CAFM. After t_{grant} the atom is granted access to the CAFM, and after $t_{delivery}$ the atom arrives at its destination. Then some time t_{accept} may pass until the atom is accepted by the target and finally after $t_{terminate}$ this atom will be terminated and the master gets informed about this, so it knows the transfer of this atom has been finished. Figure 4.5 shows the life cycle of such atoms, regardless of the type of the atom.

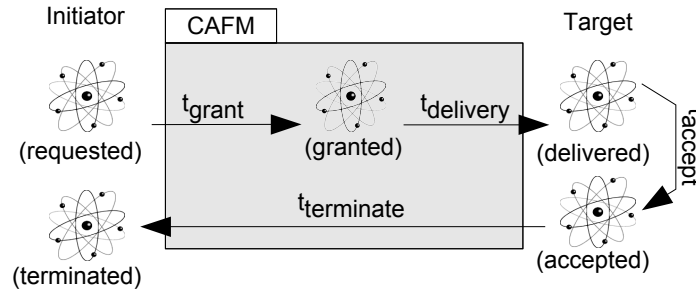


Figure 4.5: An atom's life cycle

4.5 A data representation and transport mechanism for TAQ

In the following, a SystemC implementation technique for atoms and quarks is proposed which constitutes the base of the GREENBUS fabric.

4.5.1 Data representation

A straightforward approach for the data representation of atoms would be a data container encompassing the atom state ('requested', 'granted', etc.) and all quarks that are related to the atom.

However, this approach has some drawbacks. First, there are several quarks for which it makes sense to transfer them with each atom of a transaction repeatedly. A router model for instance can be kept simple if each atom of a transaction carries the **MAddr** quark. It is used by the router to identify the right atom target port. Otherwise, more complex housekeeping would be required inside the router, it would need to store a target port \Leftrightarrow master port relation in an internal memory for each transaction. The like is true for other quarks such as **MID**, **MBurstLength**, and so on. Thus, master and slave would need to copy these ‘redundant’ quarks from atom to atom. A second drawback is that realizing atoms as data containers would imply to use pass-by-value for the atom transport. Otherwise, if using pass-by-pointer instead, each (local) state change of an atom would be immediately visible to all (other) components involved in a transaction, so that again additional housekeeping would be required to for instance compare the state of a previously sent atom with its state when it returns.

As a result, the quarks ‘carried’ by an atom should not be copied from initiator to target. Instead, they should reside in the initiator and are accessed by reference. The atom state, on the other hand, should be passed-by-value. The outcome of these considerations is to represent an atom as a

`pair<TC, PHASE>`.

TC is an access handle that provides an interface with which to access (set/get) the quarks of a transaction. PHASE is a simple data structure containing an atom type identifier (‘init’, ‘data’, etc.) and its state (‘requested’, ‘granted’, etc.). A transaction is comprised of a sequence of such pairs. This is illustrated in figure 4.6.

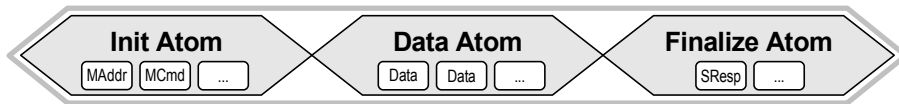


Figure 4.6: GREENBUS transaction³

All quarks that are related to a transaction are stored in a *transaction container* (TC). It can reside in the initiator of the transaction during its whole life cycle – which is the master. All other components involved in the transaction use access handles to get and set quark values.

While the TC always contains all quarks supported by GREENBUS, in high-level models only few of them are used. For setting up a PV transaction, it is sufficient to set the target address (**MAddr**), the command (**MCmd**), the burst length (**MBurstLength**), and the transaction data (**GBDataType**). When moving down in abstraction, the number of used quarks increases, such that additional protocol properties (e.g., byte enables) are taken into account.

4.5.2 Transport mechanism

To efficiently support this approach, GREENBUS must provide a low-level transport mechanism for TCs that allows to construct a set of convenience functions at little or no cost. I propose two orthogonal interfaces. First, to reflect the PV level requirement to pass entire transactions a single blocking method call:

```
b_transact(TC);
```

This is in common with ST’s TAC and OSCI TLM (see 3.3). The `b_transact` method is blocking and will return when the entire transaction is complete.

³Note that this illustration does not precisely reflect the actual implementation. Although it is a useful imagination that quarks are carried by atoms, it is not always an include relation, but sometimes a true relation (one quark / two atoms).

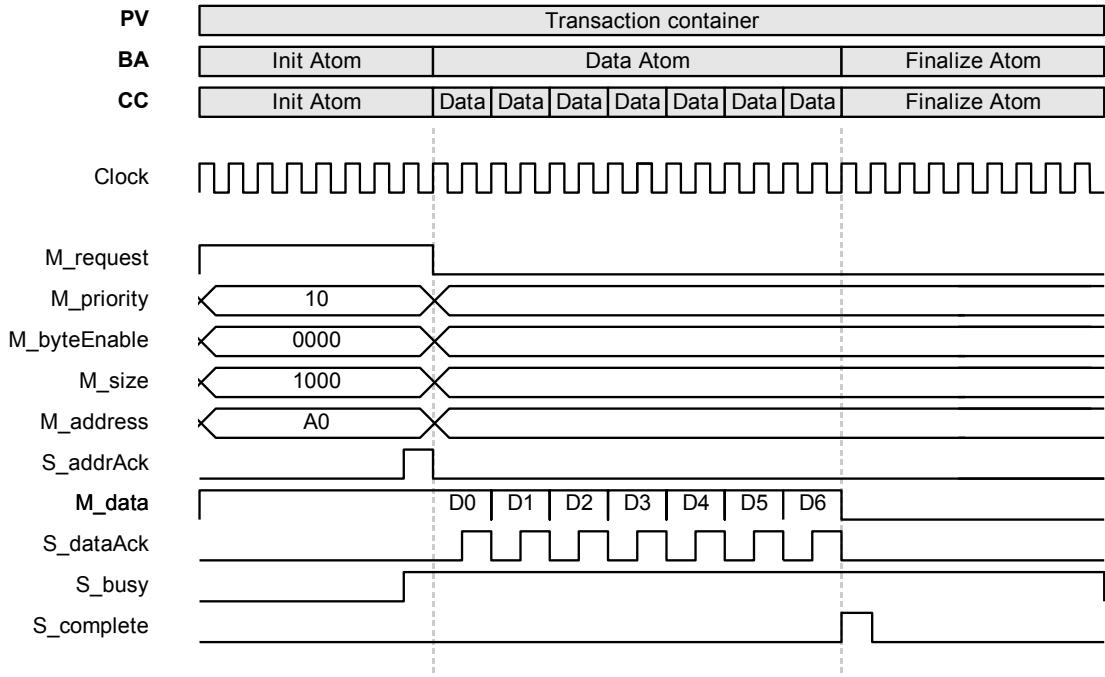


Figure 4.7: Representation of a PLB transaction with atoms and quarks

Secondly, to cover all other lower levels of abstraction, GREENBUS provides a single nonblocking method call:

```
notify(ATOM, DELAY);
```

This method call takes an atom and notifies all other (interested) components in the system of the atoms presence after the specified amount of time (**DELAY**) has passed. The function of this method is similar to that of the SystemC `notify` method for events, but with the difference, that here also a data payload (the atom) is delivered in combination with the event notification. Hence, I refer to these events as *payload events*.

When an atom is initiated, there then may be a number of events that can be generated with respect to the atom. If there are several hops (routers) between the atom initiator and the target, these will be further payload events that forward the atom to the next component on its way to the target. Furthermore, processes may be started (or resumed) to handle the atom. Finally, an atom terminate and accept event will be generated by the target port.

In GREENBUS, all events regarding to an atom life cycle can be implemented using the `notify` method call shown above, independent of the direction of the data flow (master to slave or *vice versa*). Thus, any communication in GREENBUS comes down to the two low-level transport methods. This simple interface allows both blocking and nonblocking user APIs to be constructed with little or no overhead in either case.

Example 2. To illustrate how the transaction container and the transport mechanism is used in GREENBUS, figure 4.7 shows the simulation of a RTL transaction across the IBM Processor Local Bus (PLB) with atoms and quarks⁴. Depending on the timing fidelity chosen for the bus simulation, the information carried by the transaction container is sampled in different resolution. This is pointed out at the top of fig. 4.7.

For PV accuracy it is sufficient to consider the atomic transaction, using the blocking `b_transact` method. Hence, a PV PE is only able to validate all quarks at once. Note that the illustration of

⁴Here, a PLB fixed length burst write transfer is shown. Compare [56, pg. 91].

the PV transaction in the figure is idealistic, as it suggests that the end time estimated by the PV simulation correlates with the RTL model. When there are more than one transaction taking place in parallel, this usually is not the case and the estimated time points get wrong.

For BA and CC accurate simulation, the nonblocking `notify` atom interface is used. Hence, BA and CC PEs are able to validate the quarks related to an atom at once. The `init` atom starts when the master PE requests access to the bus. It must refer to quarks that carry the corresponding signal values, such as `M_address` (target address), `M_size` (burst length), and `M_priority` (master dynamic priority). After a number of clock cycles, the request is granted by the bus arbiter and then accepted by the slave, which in the RTL model is indicated by the `S_addrAck` signal. Now the bus is allocated and a logical data link between the master and the slave has been set up. This is the time point when the request atom terminates and the data atom starts. Data transfers take place without intervention of the arbiter on the PLB until the last data word has been transferred. In the CAFM, these transfers are simulated either with a single data atom (BA transaction) or with a sequence of data atoms (CC transaction). After the last data word has been transferred, the `finalize` atom starts, indicated by the `S_complete` signal in the RTL model. The bus is still allocated until the slave informs the arbiter that it has successfully consumed the last data word by toggling the `S_busy` signal. Then the `finalize` atom terminates.

This technique enables interoperability between PV, BA, and RTL PEs. For every timing accuracy of the PLB model as defined in explanation 8, there is a representation with transactions, atoms, and quarks in GREENBUS. All abstraction levels use the same quarks for data transport and transaction set up. While timed PEs in addition will use protocol-timing specific quarks, untimed PEs may just ignore them.

The example shows that with TAQ, adapters between different abstraction levels are avoided. For example, a user API (cp. fig. 4.1 on page 35) for a PV PE that receives a PLB BA transaction just needs to wait for the `finalize` atom and then presents the whole transaction container to its PE. Note that this works independent of the bus protocol used – any transaction that adheres to the TAQ scheme and uses the standard quarks will be successfully received by any user API. There may be compatibility issues if a protocol implementation requires additional user quarks for vital protocol functions – this will be discussed in chapter 4.8.

The disadvantage of the TAQ approach is that data structure members which are timing specific are present (but unused) in untimed models. Hence, more memory may be allocated for a transaction than necessary. Chapter 4.10.1 presents a solution to overcome this issue with memory pooling.

4.5.3 Modeling communication delays

The low-level transport interface can be used by both PEs and interconnect components to model:

- latencies due to communication data processing, and
- communication delays due to protocol timing and bus arbitration.

Explicit `wait` calls in master, slave, and router processes are avoided. This has several advantages over the approaches presented in the related work:

- *Modeling efficiency:* Transactions can be ‘sent out’ and received nonblocking, so that the related processes do not ‘miss’ any (potentially important) model activity while performing communication (e.g., an interrupt request). Contrary to the nonblocking APIs in the related work, payload events deliver the transaction data to the target in the same way than blocking method calls (as a method parameter). This simplifies the development of user APIs to the GREENBUS low-level transport interface, because the number of variables and internal events in PEs can be reduced and explicit state machines using several parallel `SC_METHOD` processes are not necessary.

- *Performance modeling*: Communication data processing latencies can be easily modeled using the payload event delay parameter. For example, a slave performance model, which ‘knows’ that processing of a received data atom will take a certain amount of time can immediately finalize the atom, passing the estimated processing time as payload event delay parameter. Such a slave can suspend at the same simulation time point when it was activated.
- *Avoid polling*: Using payload events, PEs do not need to poll for communication activity (e.g., check bus state after a `wait` call has returned, listen for bus state change events, etc.). Also, they do not need to copy the transaction data from the bus using a `getData` or similar function call. Slaves can be modeled as fully passive models. They only get activated upon bus activity that is related to them.

The different delays that can be modeled with the low-level transport interface are shown in figure 4.8. From the side of the master, we can define:

- *master overhead*: overheads prior to or during data transmission, such as a busy master, or data shaping (packetizing, de-packetizing) in the send buffer;
- *transmission delay*: data transmission between master and slave. From the viewpoint of the master, it includes the transmission delay of the interconnect as well as the slave overhead.

The interconnect (i.e. routers, CAFMs, bridges, etc.) may model the following delays:

- *arbitration delay*: time that passes between a master request and permission to channel access.
- *transmission and congestion delay*: time that passes during data transport. It depends on channel bandwidth and transaction data size, and may also include extra time due to interconnect components such as bridges, as well as protocol-specific delays (e.g. due to burst slicing).

Finally, from the side of the slave, we can define:

- *slave overhead*: time that passes in the slave prior to or during data transmission, similarly to master overhead.
- *transmission delay*: data transmission between master and slave. Contrary to the transmission delay seen by the master, the transmission delay noted by the slave is shorter as it does not include the slave overhead.

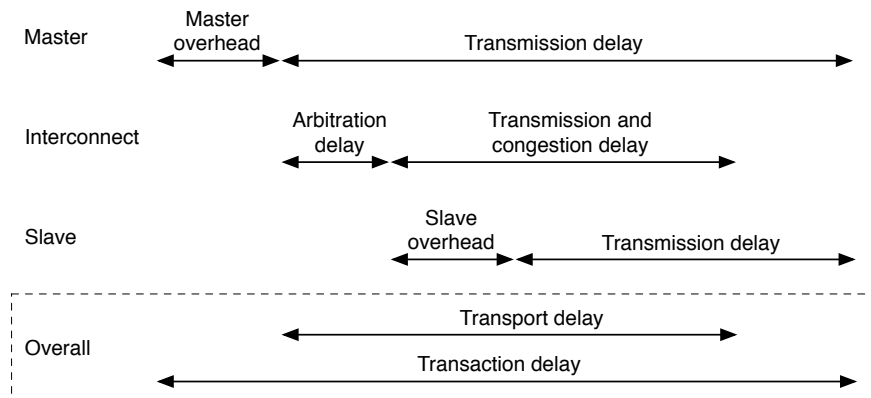


Figure 4.8: Delay model using the GREENBUS low-level transport interface

In total, there are two overall delays to a transaction:

- *transport delay* is the delay that emerges from the communication fabric;
- *transaction delay* is the total time a transaction takes.

All delays shown in figure 4.8 can be modeled using the nonblocking `notify` transport interface, which results in a BA model. The overall transaction delay can also be modeled using `wait` with the blocking `b_transact`, which results in a PV model.

4.5.4 Copy at slave

In contrast to most related frameworks, GREENBUS uses pass-by-pointer instead of pass-by-value. This means that the master is responsible for allocating the transaction container structure. This structure must be kept ‘live’ till the end of the transaction. Nobody else need allocate anything.

This scheme allows to use a single data structure with as much timing and data ‘granularity’ as required. The TC is passed to the slave by pointer – either using `b_transact` or `notify`. Hence, the slave port always receives the complete TC. However, the slave is only interested in some of the data, which may also depend on the timing point of the transaction. Thus, the slave port must *copy* the corresponding data out of the TC, as this data will not be available after the transaction has finished. We refer to this approach as *copy at slave*.

This approach has two advantages over pass-by-value:

- all components involved in a transaction can see *all* information about this transaction during its complete lifetime. They can also attach *private* information to the TC which they can read out when they see this TC again. For example, a router can annotate the target port, such that it only needs to resolve it once for a multi-atom transaction.
- as all slave ports are forced to make a copy of important transaction data, pass-by-pointer can be used safely in the TLM fabric. By using *shared pointers*⁵ TCs in GREENBUS get automatically deleted when they expire. The user needs not to care about memory management.

4.6 Connection layer

4.6.1 Basic port

Internally, all GREENBUS ports use the low-level blocking `transport` and nonblocking `notify` methods as basic means of communication. They are implemented in the GREENBUS *basic port* (class `basic_port`, see file `tlm_port.h`). The higher-level master, slave, initiator, and target ports are hierarchically constructed from this port.

The basic port contains two SystemC ports for each of the two interfaces (blocking and nonblocking), an `sc_port` and an `sc_export`. The `sc_ports` are used to do `b_transact` and `notify` interface method calls on another (connected) basic port. The `sc_exports` are used to export an implementation of the interface methods to the other basic port. Two basic ports can be connected together using the operator `()`:

```
basic_port_a(basic_port_b);
```

Figure 4.9 depicts the nonblocking communication interface. On the left-hand port A, a PE calls `notify` with an atom and a notification delay. As a result, a `notify` interface method call takes place on the right-hand port B via the `sc_port`-to-`sc_export` connection. The `notify` interface method is implemented by a special channel that processes payload events, the *payload event queue* (PEQ). Its interface is exported to port A by the `sc_export` of port B. The advantage of having an `sc_export`

⁵See BOOST smart pointers [2].

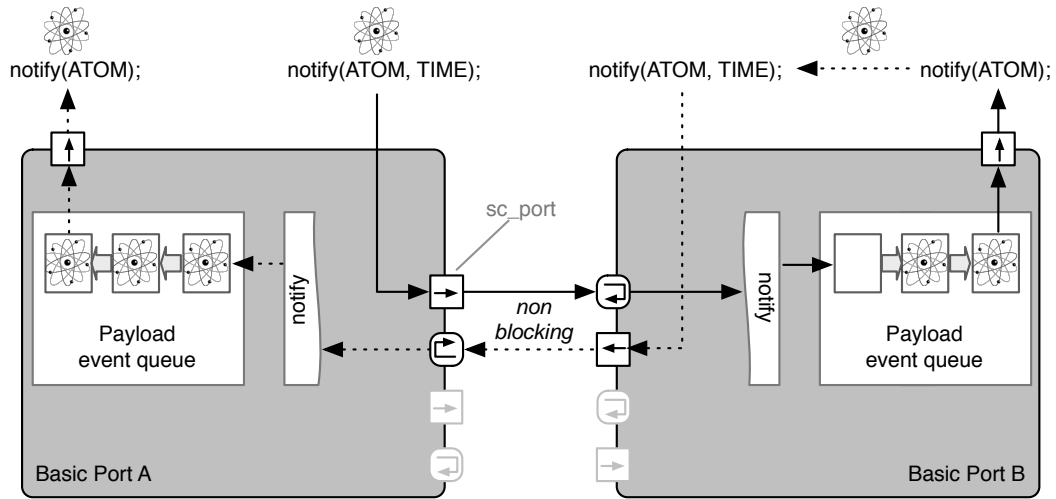


Figure 4.9: Nonblocking basic port communication

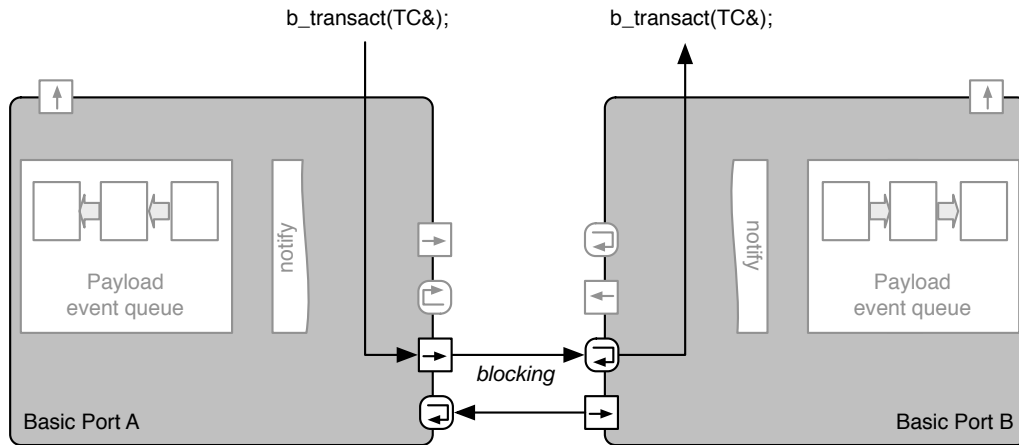


Figure 4.10: Blocking TLM port communication

is that the `notify` implementation is separated from the basic port implementation and thus can be easily replaced, e.g., by a PEQ with debug features.

After the notification delay has passed, the PEQ delivers the atom by calling `notify` in the target PE. The receiver of the atom now may alter its state and uses the same transport mechanism to pass it back to the source. This is indicated by the dotted arrows in figure 4.9. On its way back, the atom passes the PEQ of port A.

Figure 4.10 shows how blocking communication over the basic port works. In contrast to the nonblocking interface the blocking interface is untimed and therefore does not require the event queue. The `b_transact` method call is immediately forwarded to the target, again using `sc_port-to-sc_export` binding. Here, the `sc_export` exports the interface of the target module. It must implement the `b_transact` method that is defined in the `tlm_b_if.h`). Again, this interface can be used in both directions.

4.6.2 Payload event queue

The PEQ implements the `payload_event_queue_if` interface shown in listing 4.2. This interface supports several ways of firing a payload event with a given notification delay.

Listing 4.2: Payload event queue interface (payload_event_queue_if)

```

1 template<typename PAYLOAD>
  class payload_event_queue_if : public virtual sc_interface
2 {
3 public:
4     virtual void notify (PAYLOAD& p, double when, sc_time_unit base) =0;
5     virtual void notify (PAYLOAD& p, const sc_time& when) =0;
6     virtual void notify (PAYLOAD& p, const enumPeqDelay delay=YIELD) =0;
7     virtual void cancel_all() =0;
8 };

```

When `notify` is called, the payload event is enqueued in a time ordered list. When the given delay has passed, the target gets notified and the payload is delivered. The notification of the target takes place using an `sc_port`, to which the target must be connected. The port is bound to the `payload_event_queue_output_if` interface, which defines a simple `notify` method that has to be implemented by the target (listing 4.3). The method passes the payload to the target. In GREENBUS, this payload always will be an atom, i.e. `pair<TC, PHASE>`. The PEQ implementation can be re-used for other tasks than bus modeling by specifying another payload data.

Listing 4.3: PEQ output interface (payload_event_queue_output_if)

```

1 template<typename PAYLOAD>
  class payload_event_queue_output_if : public virtual sc_interface
2 {
3 public:
4     virtual void notify (PAYLOAD& p) =0;
5 };

```

Figure 4.11 illustrates the PEQ operation with an example. An initiator creates two payload events. The first is created at simulation time point $10ns$ and is given a delay of $40ns$. The second is created at time point $20ns$ and is given a delay of $10ns$. Thus, the second payload event is enqueued *before* the first in the time ordered list, and the payload event queue first serves the second payload event and then the first one upon the target.

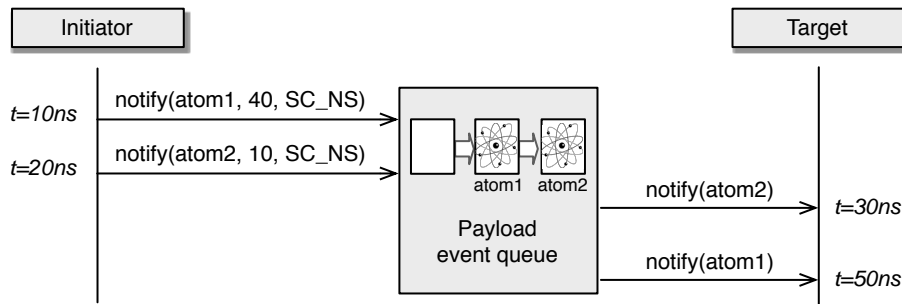


Figure 4.11: Processing of payload events

4.6.2.1 Delay types

Similar to SystemC events (class `sc_event`), the PEQ `notify` method supports the following three types of transport delay:

- *Immediate transport*: No delay is specified at all. The `notify` method in the target is called at the current delta cycle.

- *Zero-time or delta transport:* A zero time delay (`SC_ZERO_TIME`) is given. The `notify` method in the target will be called in the next delta cycle.
- *Timed transport:* A delay time greater than zero is specified. The `notify` method in the target will be called in the first delta cycle of the specified simulation time point in the future.

4.6.2.2 Immediate notifications

While there is no doubt about the semantics of zero-time and timed transport, special attention should be paid to immediate transport. In the TLM fabrics reviewed in chapter 3.3, we find two fundamentally different implementation techniques for immediate data transport:

1. *Direct method call:* This is usually found together with a blocking API. The blocking transport primitives of OSCI-TLM, ST TAC, and OCCN do a direct interface method call in the slave.
2. *Immediate event:* This is typically used with nonblocking APIs. The nonblocking transport methods of OSCI-TLM, OCP, and IBM CoreConnect trigger an immediate `sc_event`, which activates a slave process in the same delta cycle.

Though aiming at the same functionality, these two approaches result in considerably different model behaviors and transport techniques. Using direct method calls, the slave method is executed as part of the master process. It does not return until the slave has completed processing the transaction data. No separate (concurrent) thread is started. Hence, the payload data must be passed-by-value, since otherwise a master would see modified data values in its memory when the slave method returns. This is illustrated in figure 4.12a.

Using immediate event notification, however, the payload data can be passed-by-pointer. The immediate event schedules the slave process to be activated *after* the master process has suspended. Thus, the master can safely continue working with the transaction data after it has called the (non-blocking) transport method, as the slave will not start modifying the data before the master process yields. This ‘safe’ kind of immediate transport, however, is not supported by any of the blocking TLM APIs I have reviewed. This may lead to defective models if these APIs are not used with reasonable care.

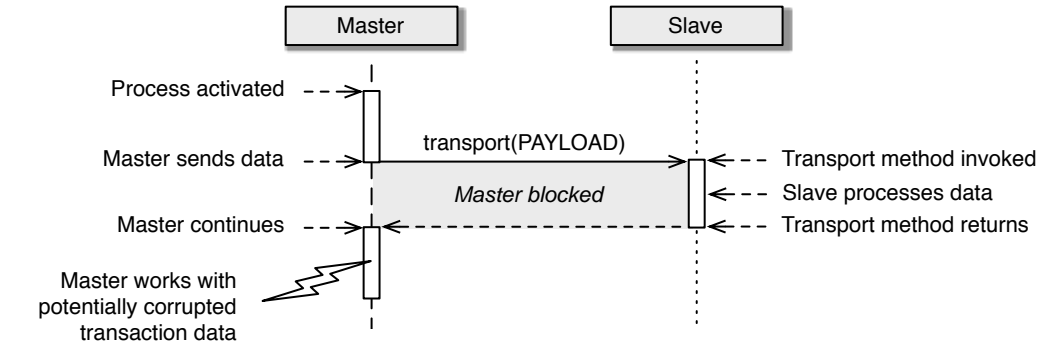
In GREENBUS, the payload event queue allows to overcome this issues in an elegant way. By using an *internal* immediate `sc_event` notification, the PEQ enables safe immediate payload transfers using pass-by-pointer. When `notify` is called without the delay parameter, the internal `sc_event` is used to call the slave `notify` method *at the end of the current delta cycle*. Hence, the slave is not activated before the master process has been terminated (or suspended, in case of an `SC_THREAD`). This immediate ‘yield’ notification is illustrated in figure 4.12b.

Note that in both cases no simulation time passes. Both blocking and nonblocking user APIs can be built with this transport mechanism, and blocking and nonblocking transport method calls can be mixed for processing different atoms of the same transaction.

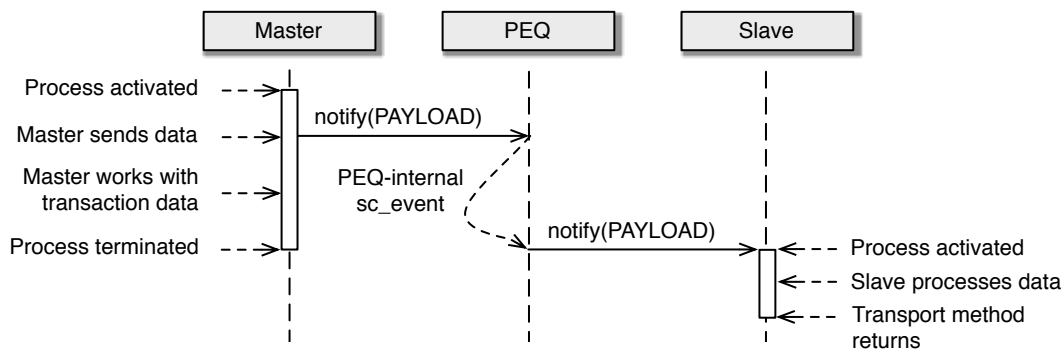
Contrary to these considerations, in special cases it may make sense to sidestep the PEQ-internal event. In abstract models that aim at rough execution time estimations every single additional event has an impact on the overall simulation performance. Random data may be used where data integrity is not an issue. The payload event queue provides the special delay type `IMMEDIATE` to support such models. That is, a call

```
notify(PAYLOAD, IMMEDIATE);
```

on the PEQ will result in a direct `notify` interface method call on the slave.



a) Non-delayed data transport using direct method calls



b) Non-delayed data transport using immediate 'yield' payload events

Figure 4.12: Effects of immediate (non-delayed) data transfers

4.6.3 Hierarchical ports

The basic port enables to set up point-to-point links using the low-level transport interface. Higher-level functionality is implemented by extended ports, which are derived from basic port using the inheritance tree shown in figure 4.13. Table 4.6 summarizes the functionality of the several ports.

Two main port types are distinguished: initiator and target ports. Initiator ports are intended to be used by components that *actively* start a transaction. They therefore are the base class for master ports. Target ports are intended to be used by components that *passively* listen for transaction requests. They therefore are the base class for slave ports.

4.6.3.1 Initiator port

The initiator port extends the basic port with functionality to create and manage transaction containers. A new TC is created by a function call:

```
accessHandle ah = myInitPort.createTransaction();
```

This function returns an access handle to a pre-initialized TC. The values of the quarks carried by this container can be read and written using the generic access methods (see 4.7). A memory pool is used for the construction of TCs. Before start of simulation, the pool constructs a number of TCs and allocates memory for them. During simulation, these TCs are re-used to minimize simulation overhead due to memory allocation and de-allocation.

Table 4.6: GREENBUS ports

Port	Functionality
Basic port	Low-level transport interface (blocking <code>b_transact</code> , nonblocking <code>notify</code>)
Initiator port	Transaction container creation and memory management
Target port	Target address space administration
Generic initiator port	Generic protocol implementation for initiator access (router, adapters, other interconnect components)
Generic target port	Generic protocol implementation for target access (router, adapters, other interconnect components)
Master port	User API implementation for master PE access
Slave port	User API implementation for slave PE access

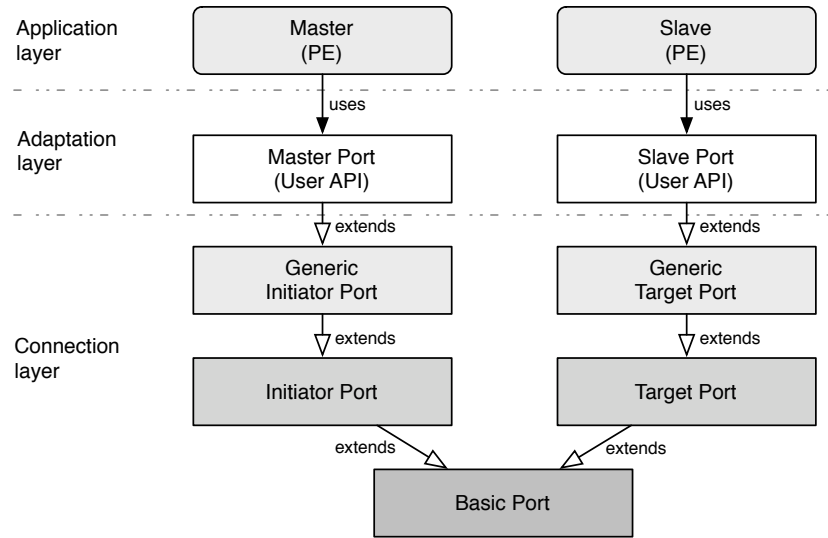


Figure 4.13: Port inheritance class diagram

Using a template parameter, initiator ports can be configured for different TC implementations. This enables model interoperability. Each port in a heterogeneous model can be configured individually to return the right TC for ‘his’ PE when `createTransaction` is called.

4.6.3.2 Target port

The target port contains two configurable parameters⁶, `base_addr` and `high_addr`. These parameters specify the target port address range. If a target port is connected to a router, they are used to build the address map in the router. The parameters can be set either at runtime or using a configuration file.

4.6.3.3 Generic initiator port and generic target port

The generic initiator port and the generic target port classes extend the initiator port and target port classes with an implementation of the *generic protocol* (GP). This protocol eases TC handling and provides convenience functions for sending and receiving atoms. It is the standard interface to the GREENBUS communication layer and is used as a base for the development of user APIs. It is addressed in chapter 4.7.

⁶Configurable parameters are explained in 6.2.3.

4.6.3.4 Generic router initiator and generic router target port

To enable multiple connections to router components, special multi-port versions of the generic ports are available, namely generic router initiator port and generic router target port. These ports can be connected to an arbitrary number of initiator or target ports respectively and contain a vector, which can be used by the router to iterate through the list of connected ports.

4.6.3.5 Master and slave ports

Finally, master and slave ports form the PE interfaces to a GREENBUS interconnect. Contrary to the above described ports, which implement the standardized generic interface, master and slave ports implement PE-specific user APIs. My GREENBUS reference implementation includes user APIs for the following interfaces:

- OSCI TLM 1.0 [109] and OSCI TLM 2.0 [90];
- OCP-tl1 and OCP-tl0 [95];
- ST TAC [119];
- SHIP and SimpleBus [69, 70, 73].

PEs designed for one of the above interfaces can be immediately connected to a GREENBUS, using the appropriate master and slave ports as a replacement for the original ports. Other user APIs can be implemented using the design patterns presented in appendix B. An approach for automated user API generation from IP interface descriptions is presented in [75].

4.6.3.6 Summary

Figure 4.14 illustrates the GREENBUS port hierarchy.

4.6.4 Concluding remarks

In summary, the communication model implemented by the GREENBUS ports contributes to the following concepts:

Pass-by-pointer / copy at slave: All data related to a transaction resides in the master port TC. A shared pointer to the TC is passed through the interconnect. Simulation overhead is decreased in comparison to pass-by-value approaches.

Automatic memory management: The transaction data structures are created and managed by the TLM fabric, not by the PEs. Thus, PE programmers do not need to take care about memory management.

Mixed abstraction modeling: Due to the separate blocking and nonblocking transport interfaces PEs can be developed that use a fast ‘performance mode’ when receiving PV transactions and switch to a slower ‘accuracy mode’ to process BA and CC transactions. The source code for both operation modes is separated.

Reduce housekeeping: Ports, routers, and other components can attach protocol simulation and other meta information to a TC while it travels through the interconnect. When it is seen again (next atom), the attached meta information can be re-extracted.

Ease instrumentation and analysis: As all communication uses the same standardized low-level transport interfaces, analysis and debug features can be implemented that work independent of the used CAFMs and user APIs.

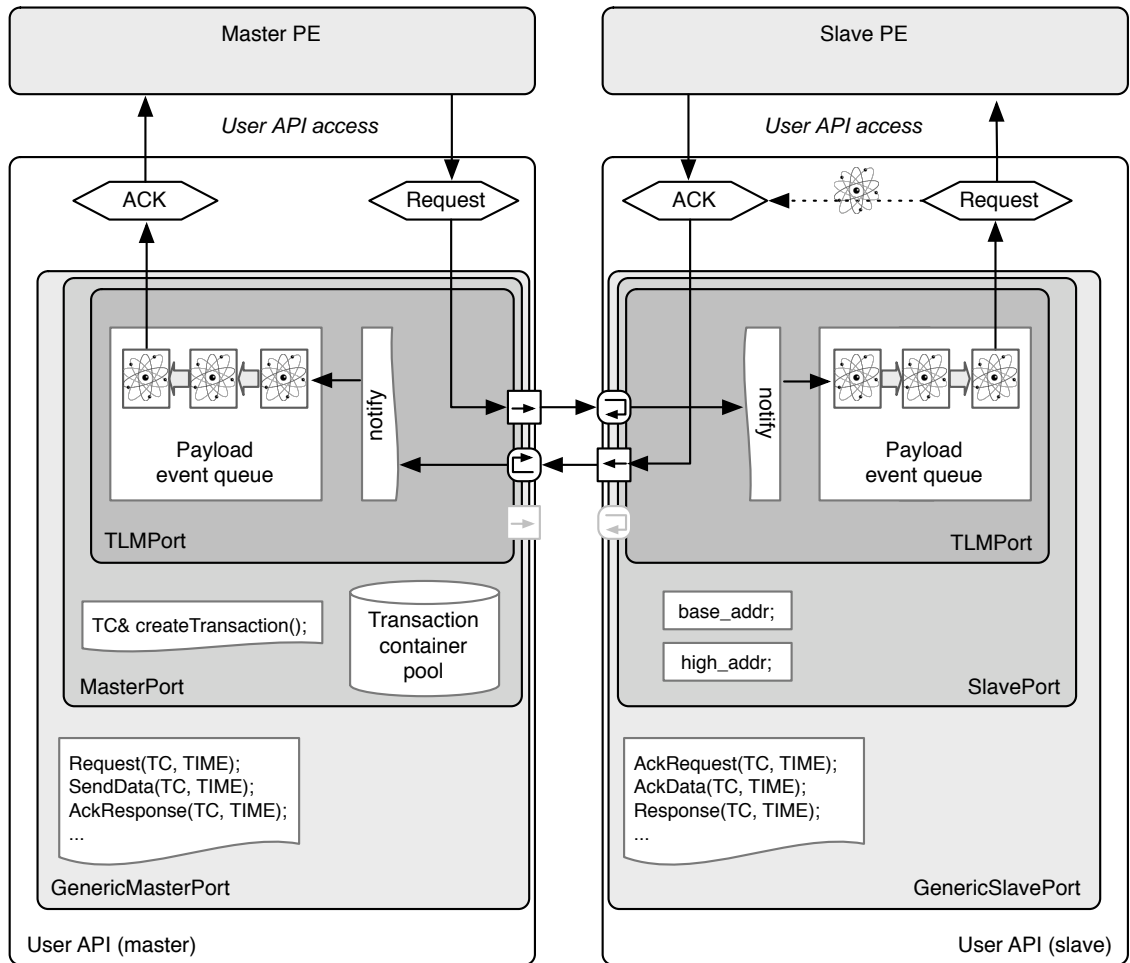


Figure 4.14: Master-slave communication via hierarchical GREENBUS ports

4.7 Generic protocol

While the low-level transport interface in combination with the transaction container is an efficient base for communication modeling, it is not very convenient nor safe in use, and it does not implicitly guarantee interoperability. If the blocking `b_transact` and nonblocking `notify` interface methods are utilized improperly, transactions may fail or unexpected behavior may occur. For example, a master should be prevented from starting a new transaction with a finalize atom, a slave should not send a response without having received a request before, and certain quark values should be protected from overwriting while an atom is in progress.

To avoid such problems and ease communication modeling, the generic ports implement a generic protocol, which is the standard ‘user API to GREENBUS’ interface. It splits up into two parts:

1. Transaction container access;
2. Atom transport.

The functionality of the GP is distributed over a set of interface methods. They are implemented by the generic initiator port (master access methods), the generic target port (slave access methods), and the multi-port versions of these ports (router access methods). Many problems due to wrong API usage are avoided a priori, since masters simply do not have access to interface methods that are intended for router and/or slave use only. Furthermore, an important advantage is that different versions of the access methods can be implemented:

1. *Generic protocol implementation for performance simulation*

This implementation of the generic access methods aims at fast design space exploration. As few runtime checks as possible are implemented and time consuming operations such as dynamic casts and memory copies are avoided. Method inlining and pointers are used to optimize simulation performance.

2. *Generic protocol implementation for debug and analysis*

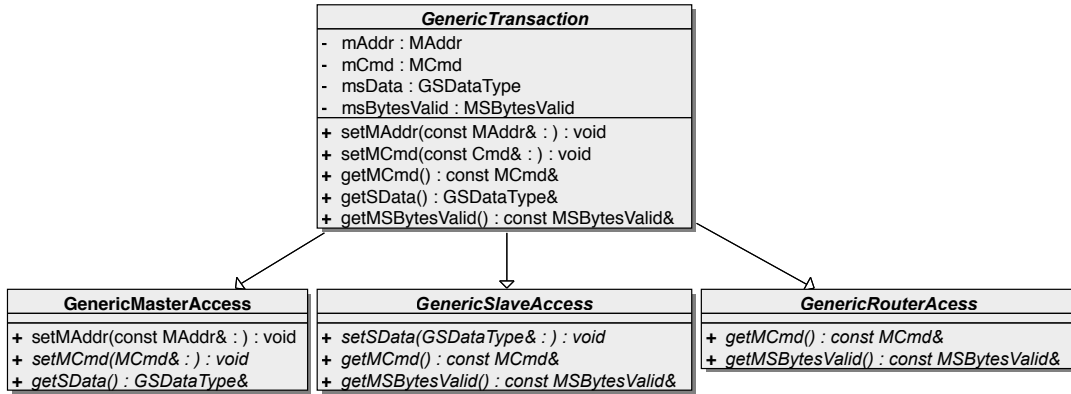
This implementation of the generic access methods includes assertions, protocol rule checks, and log and warning messages. This eases debugging and helps discover modeling errors. All interface methods are instrumented to enable transaction recording and data introspection. In combination with a debug tool, they can be used for interactive communication analysis (see 6.3).

4.7.1 Transaction container access

Figure 4.15 shows a possible realization of the TC in the GP implementation. All quarks related to a transaction are stored in a single data container, the `GenericTransaction`. The access methods are defined by the virtual interfaces `GenericMasterAccess`, `GenericSlaveAccess`, and `GenericRouterAccess`. Their implementation resides in `GenericTransaction`. Using this hierarchy, the `createTransaction` method in the generic initiator port can create a `GenericTransaction` but return a `GenericMasterAccess` handle. Thus, a master has only access to the interface methods defined in `GenericMasterAccess`. Similarly, the generic target port delivers a `GenericSlaveAccess` upon reception of a `GenericTransaction`, and routers work with `GenericRouterAccess` handles.

The generic access methods set and get quark values. For example, the `setMAddr` method sets the value of the address quark in the TC, and with `getMAddr` it can be read out again. While most access methods are related to exactly one quark, others affect multiple quarks at once. Note that for the user this does not matter as long as the expected behavior is achieved.

⁷This figure shows only a selection of interface methods to keep the class diagram simple. Please refer to the source code documentation for a full list.

Figure 4.15: Possible implementation of the generic transaction container⁷

To give an example, the following code sets up the TC for a write transaction:

```

accessHandle ah = myInitPort.createTransaction();
th->setMAddr(0x2000); // set target address
th->setMCmd(Generic_MCMD_WR); // write command
th->setMData("Guten Morgen Braunschweig!"); // set transaction data

```

The result will be a single-request multiple-data burst. The code in the slave port to receive this transaction could look like this:

```

switch(th->getMCmd()) {
case Generic_MCMD_WR:
    processWrite(th->getMData(), th->getMBurstLength());
    ...
}

```

Using method overloading, several access methods with the same name are provided for different parameter data types, e.g.:

```

th->setMData(uint64); // set single-beat data word
th->setMData(std::vector<byte>&); // set burst data vector

```

Both methods set the `MData` quark, thereby translating the user data types into the data type used in the TC. In addition, they also set the `MBurstLength` and `MBurstPrecise` quarks, as this information is implicitly passed with the method call as well.

4.7.1.1 User data

The data types supported for the read and write data of transactions differ significantly in today's TLM fabrics. As GREENBUS aims at interoperability, it must provide a universal data type that can be applied for both approaches. The `GBDataType` therefore supports two modes of operation:

- *Pointer mode:* A pointer (`void*`) to a user object is transported.
- *Byte stream mode:* A vector of bytes (`std::vector<byte>`) is transported.

The byte stream mode enables data exchange between PEs that use incompatible user data types. The user data types are serialized into a stream of bytes in the sender and are de-serialized in the receiver. Several overloaded `setData` methods are available to create a byte stream representation of standard data types, such as `char`, `uchar`, `int`, `uint`, `long`, `ulong`, `std::string`, etc. With similarly overloaded `getData` methods, these data types can be reconstructed again.

Listing 4.4: Serializable interface

```

class gb_serializable_if {
2 public:
  /// serialize this object into a byte vector
4  virtual const gs_uint32 serialize(GBDataType &data) =0;
  /// reconstruct this object from a byte vector
6  virtual const gs_uint32 deserialize(GBDataType &data) =0;
  /// get the serialized size of this object
8  virtual const gs_uint32 getSerialLength() =0;
};

```

For user data types, serialization and deserialization must be implemented by the user. A `gb_serializable_if` interface is provided with GREENBUS for that purpose. It is shown in listing 4.4. Objects that implement this interface can be converted into a byte stream representation and back. This is used in GREENBUS to enable data exchange between PEs that use different data representations.

4.7.2 Generic communication

Mapping user interface data structures to GREENBUS quarks is necessary but not sufficient to enable interoperation of heterogeneous PE. It ensures a common semantic understanding of the transferred data values. The second matter to be decided is the protocol. All ports involved in a transaction need to implement the same protocol in order to establish a transaction. In other words, the sequence of atoms sent back and forth between master and slave ports during a transaction must follow rules for reliable communication, with both user APIs being sure to not talk at cross-purposes.

The GP is a simple request-response protocol that enables read and write transactions of variable length, using the following rule to build transactions out of atoms:

Init → Data* → Finalize

All transactions start with an init atom, followed by any number of data atoms or none, and are terminated by a finalize atom. If there is only one data atom, it represents both data and finalize atom at the same time. Initiator and target ports provide a number of generic transport methods to enable straightforward communication modeling using the GP. They are listed in table 4.7.

Table 4.7: Generic protocol API

Method	Description	Atom type
Generic initiator port API		
Request	Issue read / write request	Init
SendData	Send read / write data	Data / Finalize
AckResponse	Acknowledge read data reception	Data / Finalize
ErrorResponse	Deny read data reception / signal error	Data / Finalize
Idle	Master is idle (for optional use)	-
Generic target port API		
AckRequest	Acknowledge master request	Init
ErrorRequest	Deny master request / signal error	Init
AckData	Acknowledge write data reception	Data / Finalize
ErrorData	Deny write data reception / signal error	Data / Finalize
Response	Send read data to master	Data / Finalize
Idle	Slave is idle (for optional use)	-

The methods take a transaction container and an optional delay time. Each of them supports the following kinds of invocation (here shown using the `Request` method as an example):

```
Request(transactionHandle th);
Request(transactionHandle th, const sc_time &d);
Request(transactionHandle th, double d, sc_time_unit u);
Request(transactionHandle th, enumPeqDelay d);
```

All generic transport methods create an atom and send it using the basic port `notify` method. The optional delay value is directly passed to `notify`.

The generic transport methods support both nonblocking and blocking communication. The default behavior is nonblocking. That is, the method returns immediately after issuing the appropriate atom.

Blocking behavior can be achieved by adding the suffix `.block` to the method call:

```
myPort.Request.block(th, 20, SC_NS);
```

In blocking mode, the methods suspend the calling process until the corresponding reply atom has been received. The method listens for incoming payload events on the port and compares each received atom with the atom that has been sent out. If the TCs referred to by the two atoms are identical, the method returns. The return value is the phase identifier of the received atom.

Each atom is accompanied by a protocol phase identifier. It is an instance of the class `GenericPhase`, which provides the list of methods shown in listing 4.5.

Listing 4.5: `GenericPhase` methods

```
1 class GenericPhase // "Phases for the Generic Protocol";
2 {
3 public:
4     /// the phases of the generic protocol
5     enum {
6         Idle = 0,
7         RequestValid, RequestAccepted, RequestError,
8         DataValid, DataAccepted, DataError,
9         ResponseValid, ResponseAccepted, ResponseError,
10        LAST_GENERIC_PHASE
11    };
12
13    gs_uint32 state;
14
15    /// methods to query the phase state
16    bool isRequestValid();
17    bool isRequestAccepted();
18    bool isRequestError();
19    bool isDataValid();
20    bool isDataAccepted();
21    bool isDataError();
22    bool isResponseValid();
23    bool isResponseAccepted();
24    bool isResponseError();
25
26    ... (constructors and further attributes) ...
27 };
```

The phase identifier is created with the init atom of a transaction (by calling one of the `Request` methods) and is then passed as the second argument to all subsequent generic transport method calls. With each atom of a transaction, the state of the phase identifier changes. Thus, the phase identifier keeps track of the transaction progress. Listing 4.6 shows the code in a master to send of a write transaction with two data atoms in a row. The phase identifier is passed back and forth between master and slave.

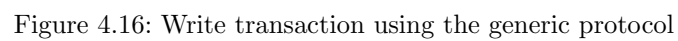


Figure 4.17: Read transaction using the generic protocol

Listing 4.6: Method call sequence in the master for sending a write transaction with one request and two data atoms, using the blocking GP interface

```

1  GenericPhase ph = myInitPort.Request.block(th);
   ph = myInitPort.SendData.block(th, ph, 100, SC_NS);
3  ph = myInitPort.SendData.block(th, ph, 50, SC_NS);

```

The generic protocol defines how the receiver of an atom must react. It must always reply with a response, being either an acknowledge or error signal, or a data atom. The GP for read and write transactions is described by the activity diagrams presented in figures 4.16 and 4.17.

Transactions that adhere to the GP are called **generic transactions**. The GP supports both single beat as well as SRMD and MRMD transfers. The minimum number of `notify` calls per generic transaction is four, for a direct connection of two generic ports. For a successful BA write transaction, the method call sequence is:

```

InitPort.Request → TargetPort.AckRequest
→ InitPort.SendData → TargetPort.AckData

```

For a successful BA read transaction, the method call sequence is:

```

InitPort.Request → TargetPort.AckRequest
→ TargetPort.Response → InitPort.AckResponse

```

4.8 GreenBus extensions and interoperability

A key requirement for GREENBUS is flexibility. For the transaction container we therefore postulated that it must be easy to extend. Similarly, the generic protocol must allow to add new phases. Figure 4.18 shows a typical TLM model as constructed by GREENBUS users such as Intel Corp. (cp. [114]). Three different TC implementations are used in this design: **GenericTransaction**, **PLBTransaction**, and **PCIETransaction**. The latter two implement user extensions.

Basically, adding new quarks and phases is not a big technical challenge. The designer can simply create a new ‘MyTransaction’ class that inherits from **GenericTransaction** and appends user quarks. The difficulty, however, is in the interoperability. How can a designer make sure that his extension

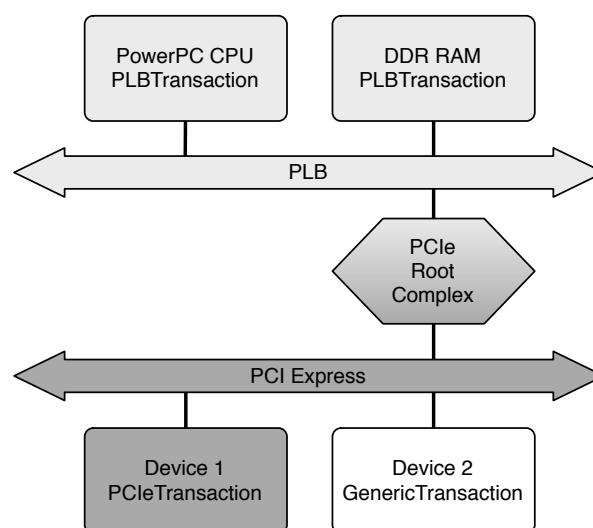


Figure 4.18: Interoperability example: mixing three transaction container types in one model

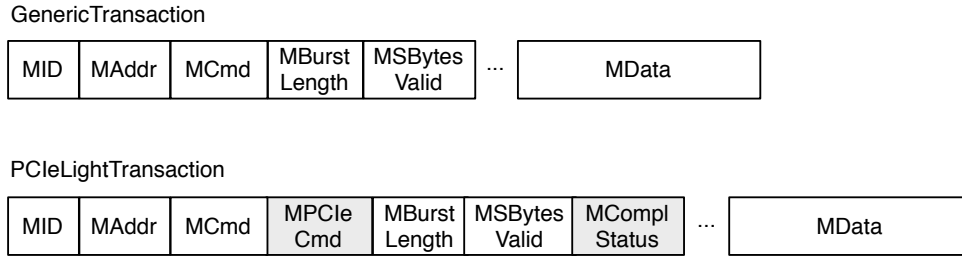


Figure 4.19: Class `PCIeLightTransaction` inherits from `GenericTransaction` and extends it by two new quarks

is still interoperable with other IP? Two aspects need to be considered to answer this question: the technical details of the extension mechanism and the compatibility level required.

4.8.1 Extension mechanism

Consider listing 4.7. The code shows a stripped-down version of our PCI Express transaction container for GREENBUS (which actually is more comprehensive, see [114]). The class defines a new command set and adds two new quarks, one for the specialized command and one for signaling a completion status (fig. 4.19). They are accompanied by four new access methods.

PEs designed for a `PCIeLightTransaction` interface will use these methods along with the methods inherited from `GenericTransaction` to create and perform transactions, which will work just fine as long as all master and slave ports (and routers) involved into such transactions do this concertedly. If, however, there are also models that use pure `GenericTransaction` containers instead, there are two obvious approaches how interoperability can be achieved: using an adapter, or making use of virtual inheritance.

The first, using an adapter, is the common approach proposed in the related work. In our example, let's assume that `MemWrite` and `MemRead` transactions are similar to generic reads and writes, whereas for the other new commands introduced by `PCIeLightTransaction` there is no direct mapping. Hence, upon reception of a `PCIeLightTransaction`, a proper adapter would check the `mPCleCmd` quark and, in case of a `MemWrite` or `MemRead`, create a new `GenericTransaction` and set its `MCmd` to either `Generic_MCMD_WR` or `Generic_MCMD_RD`. The completion status would be ignored, as there is no matchable quark in the generic transaction. The opposite direction, `GenericTransaction` \Rightarrow `PCIeLightTransaction`, would function the other way around. Here, the completion status can be calculated from the `MBurstLength` and `MSBytesValid` quarks (note that this is only possible for memory reads and writes on the PCIe bus).

4.8.1.1 Dynamic casts using virtual inheritance

The second approach seems to be more elegant: we can utilize the C++ feature to dynamically cast 'up' and 'down' between derived class and base class to achieve interoperability. If we declare the setters and getters in `GenericTransaction` 'virtual', the compiler will generate a virtual method table (vtable, see [125, chapter 2.5.5]), so that a generic PE that receives a `PCIeLightTransaction` can simply treat it as a `GenericTransaction`, thereby unconsciously using the `PCIeLightTransaction` access methods thanks to the vtable. Thus, `PCIeLightTransaction` can overwrite the `setMCmd` and `getMCmd` methods to set `mPCleCmd` to the correct value.

No explicit adapters are required. Figure 4.20 illustrates the resulting class hierarchy. In the other direction, PCIe PEs that receive a `GenericTransaction` instead of a `PCIeLightTransaction` can detect this by trying a

Listing 4.7: Transaction extension example

```
1  ATTRIBUTE (MMyCmd, gs_uint32);
   ATTRIBUTE (MCompletionStatus, gs_uint32);
3
   /// Enumeration for user quark mMyCmd
5  enum PCIECmd {
       NO_CMD = 0,
7     MemWrite,
       MemRead,
9     IOWrite,
       IORead,
11    CnfgWriteTy0,
       CnfgWriteTy1,
13    Msg,
       MsgWithData,
15    MemReadLocked
   };
17
   /// Enumeration for user quark mComplStatus
19  enum CompletionStatus {
       NO_COMPL_STATUS,
21    SuccessfulCompletion,
       UnsupportedRequest,
23    ConfigurationRequestRetryStatus,
       CompleterAbort
25  };

27  class PCIeLightTransaction
       : public virtual PCIeLightMasterAccess,
29     public virtual PCIeLightTargetAccess,
       public virtual PCIeLightRouterAccess,
31     public virtual GenericTransaction
   {
33  private:
       /// user quarks
35     MPCIECmd mPCIECmd;
       MComplStatus mComplStatus;
37
   protected:
39     /// getters and setters
       inline void setMPCIECmd(const MPCIECmd& s) { mPCIECmd=s; }
41     inline const MPCIECmd& getMPCIECmd() const { return mPCIECmd; }
       inline void setMComplStatus(const MComplStatus& s)
43         { mComplStatus=s; };
       inline const MComplStatus& getMComplStatus() const
45         { return mComplStatus; }

47     /// constructor
       PCIeLightTransaction()
49         : GenericTransaction(),
           mMyCmd(NO_CMD),
51         mComplStatus(NO_COMPL_STATUS)
       {}
53  };
```

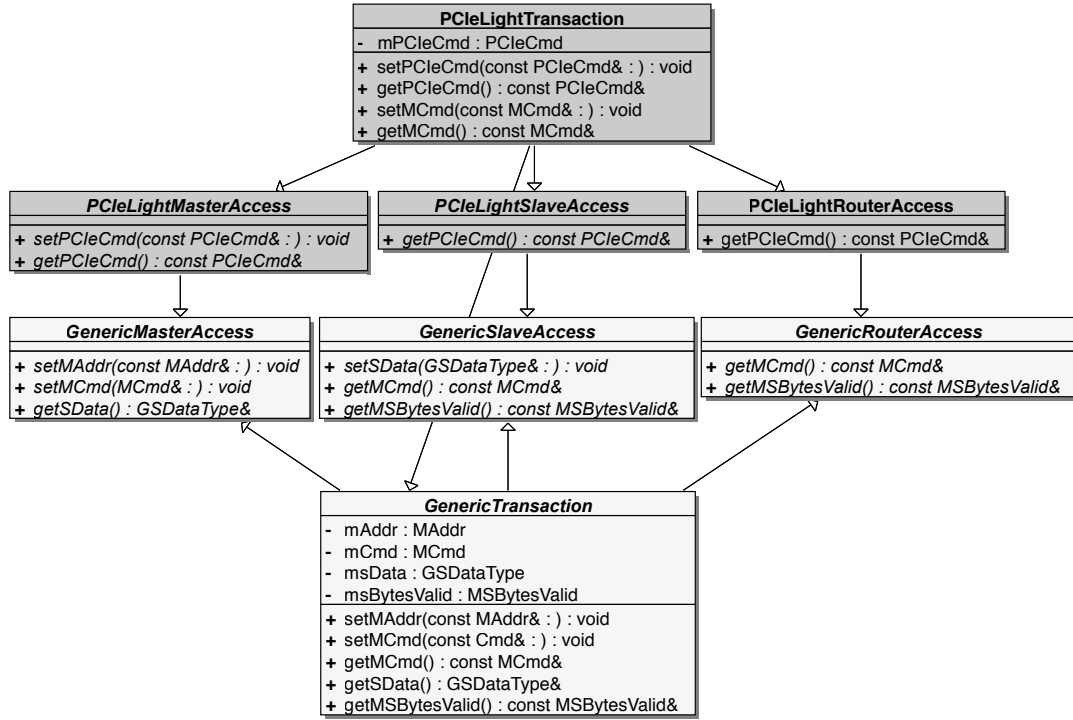


Figure 4.20: Transaction container extension using virtual inheritance

```

PCIELightMasterAccessHandle &mah =
    boost::dynamic_pointer_cast<PCIELightTransaction>(ah);

```

with `ah` being the generic access handle received from the interconnect. If the result of the dynamic cast is `NULL`, a ‘fallback mode’ for handling generic transactions can be used (the completion status will be ignored). Note that in order to achieve full interoperability for memory reads and writes, the `setMPCleCmd` method must be extended to not only set the `mPCIELightCmd` quark but also the generic `mCmd` as well.

For commands other than `MemWrite` and `MemRead` the `MCmd` quark will contain the default value `Generic_MCMD_IDLE`, which, upon reception by a generic slave, will force an error. The designer will get informed when he constructed a situation where a PCIe device sends incompatible commands to a generic PE. In that case an explicit adapter (i.e., a bridge) would be required.

The adapter-only approach has the advantage that the access methods of the generic transaction need not be overwritten in the extended transaction. The disadvantages are that an appropriate adapter must be developed for any TC extension and that these adapters slow down simulation performance because they copy quarks from one TC into another. The dynamic cast approach overcomes these disadvantages as no explicit data copying is required, but necessitates to include ‘compatibility code’ as we have seen with the `setMCmd` and `setMMyCmd` methods in the example.

Compelling as the dynamic cast approach first seems, the virtual inheritance introduces a ‘hidden’ impairment of simulation performance: each atom transfer requires a dynamic cast and each call of a TC access method causes a vtable lookup at runtime. My experiments have shown that this slows down simulation performance considerably in comparison to using a non-virtual class hierarchy. Figure 4.21 shows normalized simulation durations for different example platforms⁸ with and without usage of virtual inheritance. The performance slowdown with the dynamic cast approach is up to 50%.

⁸The example platforms can be found in the `greenbus/examples` directory of the reference implementation.

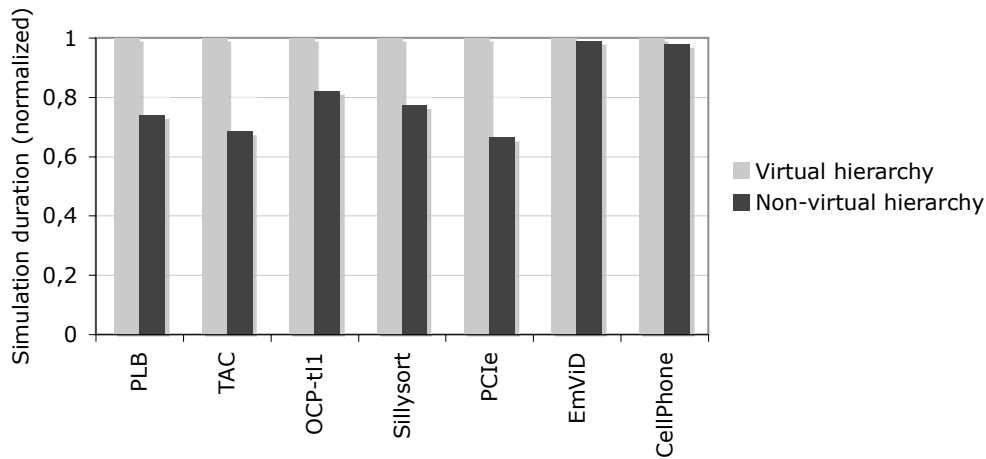


Figure 4.21: Simulation performance comparison of different demo platforms, using virtual vs. non-virtual transaction container class hierarchy

In conclusion, both the dynamic cast and the adapter-only approaches are not satisfying. While the latter is too ungainly, the former necessitates too much additional ‘legacy’ code.

4.8.1.2 The static cast alternative

A solution that emerged from intensive discussion in the GreenSocs community and also in the OSCI TLM Working Group is the static cast approach. It turns the class hierarchy upside down as shown in figure 4.22.

The generic access classes inherit from `GenericTransaction` and export its protected get and set methods with the `using` statement:

```
class GenericMasterAccess_class
: protected virtual GenericTransaction_class
{
public:
    using GenericTransaction_class::setMAddr;
    using GenericTransaction_class::setMBurstLength;
    using GenericTransaction_class::setMSBytesValid;
    ...
}
```

Again, master and slave user APIs as well as routers can only use those methods that are defined in their access classes. However, ‘upcasting’ using virtual method tables is not supported with this approach. That is, *all* master, slave, and router ports in the model that encounter PCIe TCs must be compiled with the `PCIELightTransaction` type. Otherwise, generic PEs wouldn’t use the setters and getters that are overwritten in the extended TC. So the big difference to the dynamic cast approach is that the TC type must be defined at compile time. This might appear as a disadvantage, because now every time a generic PE is connected to an ‘extended’ PE the designer must change the code of the generic port so that it is compiled with the correct extended TC type. But by making use of the C++ preprocessor, this additional effort can be completely avoided: an include file for the extended TC simply re-declares `GenericTransaction`, `GenericTargetAccess`, etc. using the `typedef` feature of C++.

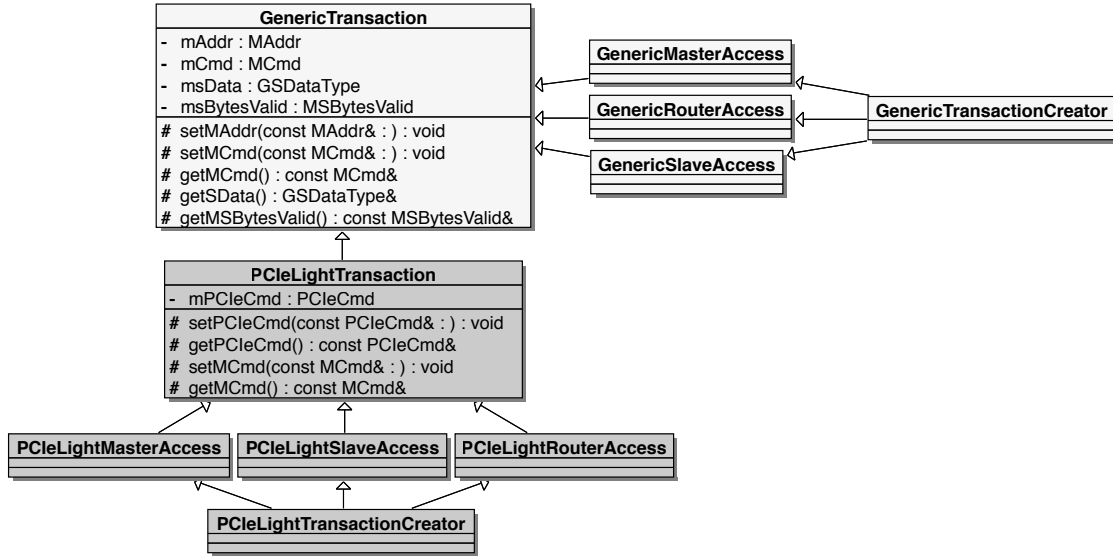


Figure 4.22: Transaction container extension using static cast hierarchy

With this approach, the example platform from figure 4.18 splits up into two subsystems: a PLB and a PCIe subsystem. The top-level file of the PLB subsystem looks like this:

```
#include "greenbus/transport/PLB/PLB.h" // load PLB extended TC
#include "powerpc.h"
#include "ddr_ram.h"

... instantiate and connect PEs ...
```

And the top-level file of the PCIe subsystem looks like this:

```
#include "greenbus/protocol/PCIE/PCIE.h" // load PCIe extended TC
#include "pcie_device1.h"
#include "pcie_device2.h"

... instantiate and connect devices and PCIe bus ...
```

The two top-level files must be compiled separately. Because of the different typedefs in the `PLB.h` and `PCIE.h` header files, each subsystem uses the right TC extensions. In particular, ‘Device 2’, which is actually a generic PE, is compiled using `PCieLightTransaction`. The overall platform is assembled by linking the two subsystem object files together and connecting the two buses by a bridge component. This component acts as an adapter and has the special characteristic that it deals with both PLB and PCIe transaction containers, using the former with its PLB-side port and the latter with its PCIe-side port.

The experimental results in figure 4.21 confirm that the static cast extension approach can give a significant simulation performance boost over the dynamic cast variant. The effects are especially drastic when a lot of quark accesses take place. This can be observed in the TAC and PCIe demo platforms, which both are communication-centric; i.e., most simulation time is spend in sending and processing transactions. An exception is the EmViD video processor model: here the processing of video image data takes center stage (appendix C).

In conclusion, the better extension technique is the static cast approach. Though it first might seem less elegant in comparison to the dynamic cast approach because it requires changes to the

code, it has outstanding performance advantages and moreover its disadvantages can be hidden with the aid of the C++ preprocessor. As a result of my experiments with both approaches GREENBUS supports both variants. The preferred mode can be chosen with the global switch `USE_STATIC_CASTS` in `greenbus/core/tlm.h`.

4.8.1.3 Phase extensions

In addition to extending the TC, new phases can be added to the generic protocol as well. This can be used to express protocol (sub-)states that are not explicitly covered by the generic phases, such as for instance ‘secondary request’ in a bus with address pipelining support, such as the PLB. The phases of the generic protocol are specified as an `enum` (see listing 4.5 on page 59). The last item, `LAST_GENERIC_PHASE`, can be used as the starting point to add new phases:

```
namespace PLB {
    enum {
        SecondaryRequest = GenericPhase::LAST_GENERIC_PHASE,
        WrPrim,
        RdPrim
    };
};
```

This code example actually shows the extended phases for our PLB implementation for GREENBUS, which is discussed in chapter 5.4.1.

4.8.2 Compatibility

User quarks of extended TCs are passed along with their generic associates from port to port through a GREENBUS interconnect. If the user APIs of two communicating ports are both aware of the meaning of the user quarks, they can make use of them to realize special protocol features that go beyond the capabilities of the generic protocol. Examples we have seen are special master commands and overlapping transactions. Further examples are priority based bus access and advanced routing schemes, such as ID based routing in a PCI interconnect.

All user APIs that adhere to the generic protocol can realize basic data exchange. This is an important advantage of GREENBUS because it enables communication among incompatible IP cores without the need for adapters. Though this flexibility comes with the price that for each new PE interface an appropriate user API must be constructed, this is necessary for each interface only once and ideally has already been done by the IP provider; instead of creating up to $\frac{n(n-1)}{2}$ adapters for the connection of n heterogeneous IPs without GREENBUS.

In general, we can distinguish three compatibility levels for two user APIs:

- *Full compatibility*: both user APIs use the generic protocol without extensions.
- *Partial compatibility*: one of the two user APIs requires user quarks to implement ‘special tricks’ but data exchange is still possible without them.
- *No compatibility*: one or both user APIs do not adhere to the generic protocol.

In the latter case, adapters are required. An interesting side effect of the GREENBUS layered approach is that IP providers can encrypt their source code but still enable the connection to different buses by supplying an appropriate user API. Such IPs can be connected to any GREENBUS interconnect since they ‘speak’ the generic protocol. For the designer, the protocol between IP core and user API doesn’t matter (fig. 4.23).

On the GreenSocs website (www.greensocs.com/GreenBus) currently a data base of GP extensions and user APIs is being established.

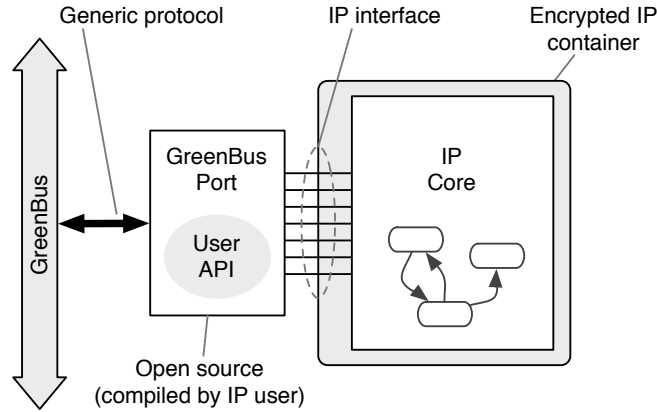


Figure 4.23: Using the generic protocol as compatibility interface for encrypted IP cores

4.9 Adaptation layer

Figure 4.24 depicts the data and control flow for interface method calls to user APIs. The picture illustrates the `write` method which is part of my implementation of ST's TAC user API for GREENBUS. The method takes three arguments: `address`, `data`, and `tac_error_reason`. It sends the data to the slave at the given address and blocks until the transaction has been completed. The master can check the value of the third argument to detect a potential transfer error.

With GREENBUS, this behavior can be realized using the generic protocol: in listing 4.8, at first a new transaction container is created and then its access methods are used to set up a write transaction. The transaction is sent by issuing a request and a data atom consecutively. For both atoms, the `.block` operator is used and the state of the reply atom is checked. If the transaction was successful, a positive TAC status is returned to the PE. The `read` method can be realized in a similar way, and when putting both methods into a class that inherits from `GenericInitiatorPort`, the result is a bus accurate (BA) TAC master port for GREENBUS.

This first example of a master user API implementation is quite simple. The data types passed to the `write` method call are natively supported by the TC. Thus, no explicit data conversion is required. Furthermore, no attempts to recover from error situations are made. For example, if the port is connected to a bus CAFM, the request atom may be refused due to bus contention. A while loop should be used to retry the request in such a case.

However, it is interesting to see that this simple implementation already supports multiple concurrent `write` method calls (as could occur in a multi-threaded master), because the `.block` operator is thread-safe. Moreover, protocol rule checking and transaction recording / analysis can be activated by simply switching to the debug version of the GREENBUS generic protocol implementation.

To complete the TAC example, listing 4.9 shows the code for the TAC write command in the slave port. The slave port implements the `notify` method of the PEQ output interface. Whenever an atom arrives, the method determines the atom type and sends the appropriate replies.

4.9.1 Implementation strategies for user APIs

The TAC example illustrated the basic principles of designing user APIs for GREENBUS. They map PE interface method calls onto the generic protocol, and vice versa. Any communication that can be mapped onto a sequence of request and response messages can be implemented this way. This requirement is fulfilled by all protocols that have been examined in this work. Thus, the generic protocol can be understood as a 'base' protocol over which other protocols can be transported. To encourage systematic user API development using the GP, appendix B gives a number of user API design patterns associated with state machine templates.

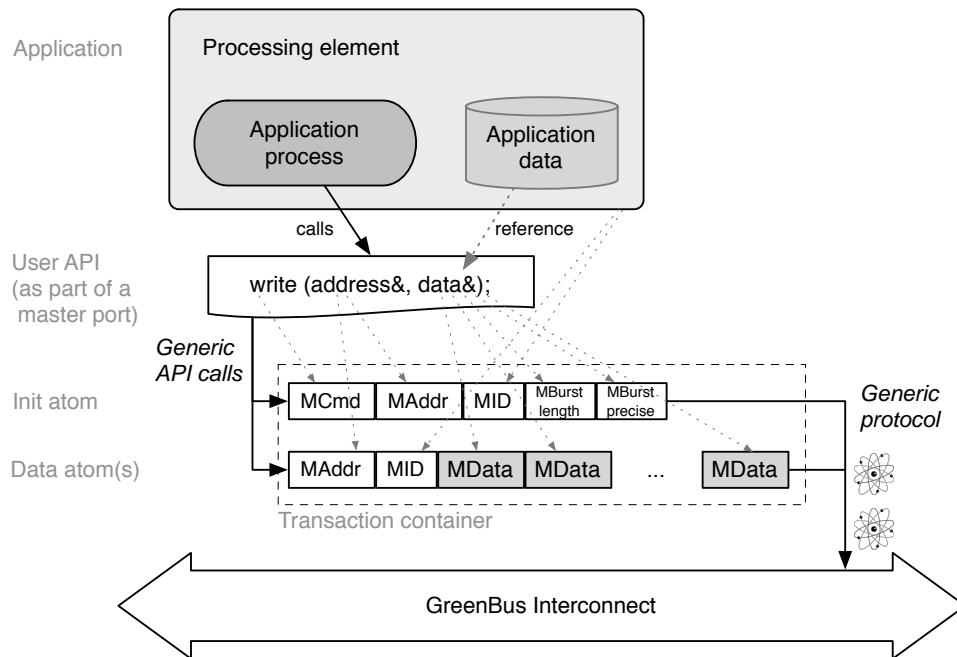


Figure 4.24: Data and control flow using the generic API and the generic protocol

Listing 4.8: TAC master port user API (write method)

```

1  tac_status write(const uint32& address,
2                  const std::vector<byte>& data,
3                  tac_error_reason& error_reason)
4  {
5      accessHandle ah = mst_port.create_transaction();
6
7      // map TAC attributes onto quarks
8      ah->setMCmd(Generic_MCMD_WR);
9      ah->setMAddr(address);
10     ah->setMData(data);
11     ah->setMSBytesValid(data.size());
12
13     // send request atom
14     GenericPhase ph;
15     ph = PORT::Request.block(ah);
16     if (ph.isRequestAccepted()) {
17         ph = PORT::SendData.block(ah, ph); // send data atom
18     }
19
20     // set TAC error state
21     tac_status status;
22     if (!ph.isDataAccepted()) {
23         status.set_error();
24         error_reason.set_reason(ph.toString());
25     }
26     else
27         status.set_ok();
28     return status;
29 }

```

Listing 4.9: Implementation of write transactions in the TAC slave port

```

1 void notify(PORT::ATOM &atom) {
    accessHandle ah = _getSlaveAccessHandle(atom); // get access handle
3   phase ph = _getPhase(atom); // get phase

5   // determine atom type
    switch(ph.state) {
7       case GenericPhase::RequestValid: // master sends request atom
            GenericMCmdType cmd = ah->getMCmd(); // get master command
9             if (cmd != Generic_MCMD_WR && cmd != Generic_MCMD_RD) {
                ah->setSError(GenericError::Generic_Error_AccessDenied);
11            PORT::ErrorRequest(ah, ph); // unknown master command
            }
13            else {
                PORT::AckRequest(ah, ph); // acknowledge request atom
15                if (cmd != Generic_MCMD_WR) {
                    // source code for other commands here ...
17                }
            }
19            break;

21        case GenericPhase::DataValid: // master sends data atom
            if (cmd != Generic_MCMD_WR)
23                SC_REPORT_ERROR(name(), "Unexpected write data phase");
            else {
25                // pass write data to slave
                if (slave_port->write(ah->getMData().getData(),
27                                    ah->getMAddr() - base_addr,
                                    ah->getMBurstLength()) {
29                    PORT::AckData(ah, ph); // acknowledge data atom
                }
31            }
            break;
33
35        default:
            SC_REPORT_WARNING(name(), "Unknown atom received");
37    }
}

```

4.10 Experiments

Starting with the first GREENBUS implementation in 2005, simulation performance has been continuously improved. In comparison to the results presented in [74], the current version is considerably faster (table 4.8). Furthermore, it is safer and easier in use. In the following some selected performance optimizations are considered. Then the simulation performance of abstract channels modeled with GREENBUS is compared to the related work. Note that the simulation accuracy and performance of GREENBUS-based CAFMs is examined in the next chapter.

Table 4.8: GREENBUS CC transactions performance with OCP-tl1 user APIs: results of the first implementation (presented in [74]) compared to the final version

Transaction size	Transactions per second			
	undelayed Ack		5 cycle delayed Ack	
	GreenBus v1	GreenBus v2	GreenBus v1	GreenBus v2
64 Byte	30,120	68,593	16,340	61,072
128 Byte	18,180	39,336	8790	33,422
2 kByte	1,410	2,756	605	2,303
5 kByte	560	1,122	240	915

4.10.1 GreenBus optimization

4.10.1.1 Transaction pooling

To minimize the overhead of transaction container creation, a memory pool is used. It contains a number of TCs that are created at initialization and then reused throughout the simulation. Thus, frequent memory allocation and deallocation for transactions is avoided, such that the number of quarks in the TC has no impact on simulation performance (cp. 4.5). Figure 4.25 graphs the performance benefit.

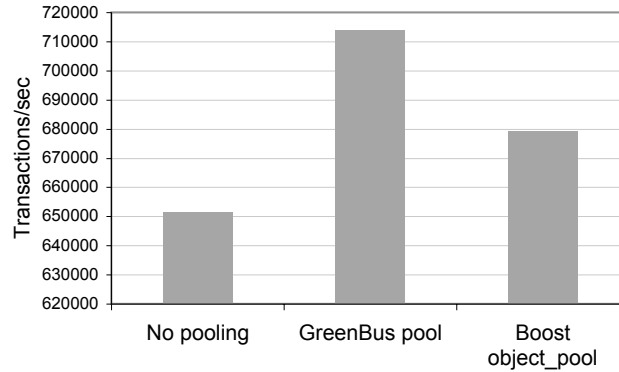


Figure 4.25: Simulation performance for point-to-point transactions via GREENBUS at BA abstraction using different transaction container creation techniques

First, we used the `object_pool` of the well-known BOOST library [107], but it turned out to be somewhat suboptimal for GREENBUS because it calls the TC constructor for each transaction and therefore re-initializes all its quarks, which is not necessary. We developed an own transaction pool that overcomes this issue and gives some additional speed.

4.10.1.2 Payload event queue optimization

My implementation of various user APIs for GREENBUS has shown that many atom transfers are either untimed (immediate notification) or zero-timed (delta cycle notification). Hence we examined whether the PEQ implementation can be optimized for these event types. We added extra event lists for immediate and delta cycle events, which work independent of the timed event queue.

Experiments with a simple bus model⁹ show a performance increase of >20% (figure 4.26). In this experiment, 1 mio. BA transactions of size 16 byte were send to a slave by a varying number of masters. The medium fill level of the PEQs in the model increases with each additional master, so this example also tests the impact of higher PEQ workload on the two implementations. It can be seen that both variants scale similarly.

4.10.1.3 SC_METHOD versus SC_THREAD

The first implementation of GREENBUS was mainly based on `SC_THREAD` processes as they are easy to code. However, experiments revealed that `SC_METHOD` processes produce less simulation overhead.

`SC_THREAD` processes possess a separate execution context in the SystemC kernel and hence a switch to the stored thread context is necessary each time a thread is started or resumed. The context switch includes saving of the current processor state and recovery of the stack and processor state of the new execution context.

⁹I used a simulation of the OPB with the generic router for these experiments, see chapter 5.1.

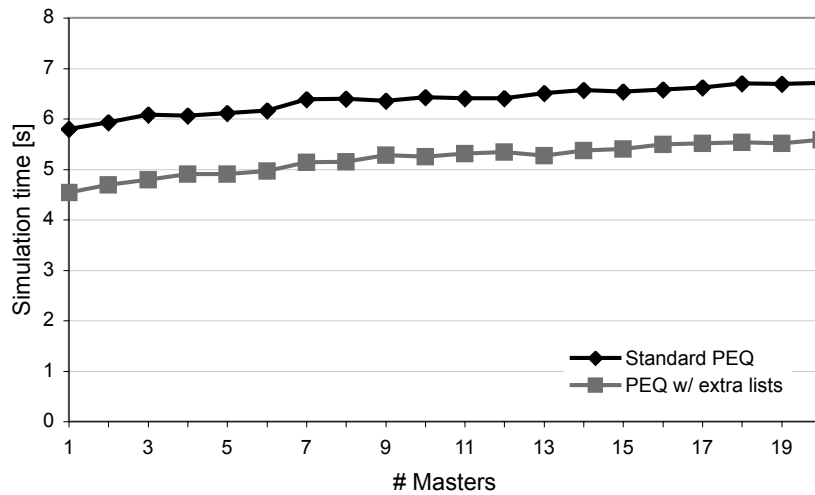


Figure 4.26: Comparison of simulation performance for 1 mio. transactions of a varying number of masters to a slave via GREENBUS at BA abstraction using different PEQ implementations

In contrast, `SC_METHOD` processes are executed in the simulation kernel context. They are simple method calls.

A conversion of all `SC_THREAD` processes into `SC_METHOD` based implementations gave a performance improvement of 21% for the CC experiment in table 4.8.

4.10.2 Simulation performance

We can measure simulation performance in different aspects:

- *Transaction size*: how performance scales with burst length.
- *Abstraction and time model*: trade-off between performance and timing fidelity of results.
- *User API and protocol*: how the fabric behavior change with different use models.

4.10.2.1 PV, BA, CC

We start with comparing the performance of GREENBUS PV, BA, and CC point-to-point transactions. For the PV and BA measurements a TAC master was connected to a TAC slave, using the appropriate GREENBUS user APIs. The TAC master produces burst writes of configurable size. The TAC slave models a DDR memory block, including realistic write latencies. For the CC measurement I replaced the TAC PEs by OCP PEs that have the same behavior but a CC accurate interface (using my OCP-tl1 user APIs for GREENBUS).

Figure 4.27 presents the results. It can be seen that simulation speed increases significantly with abstraction. PV transactions are one order of magnitude faster than BA transactions. BA transactions are one to two orders of magnitude faster than CC transactions, depending on the burst length. This confirms the TLM expectations (cp. [17, 41, 112]). Although the user APIs of the PEs are mapped onto the generic low-level protocol, the transactions can still benefit from their respective abstraction layers.

The next experiment, shown in figure 4.28, considers the impact of communication delays. For burst writes with a constant length (64 bytes) the acknowledge delay of the slave was varied. The experimental results show that no dependency between acknowledge delay and performance can be seen (variance results from operating system effects). With the payload event technique it makes no

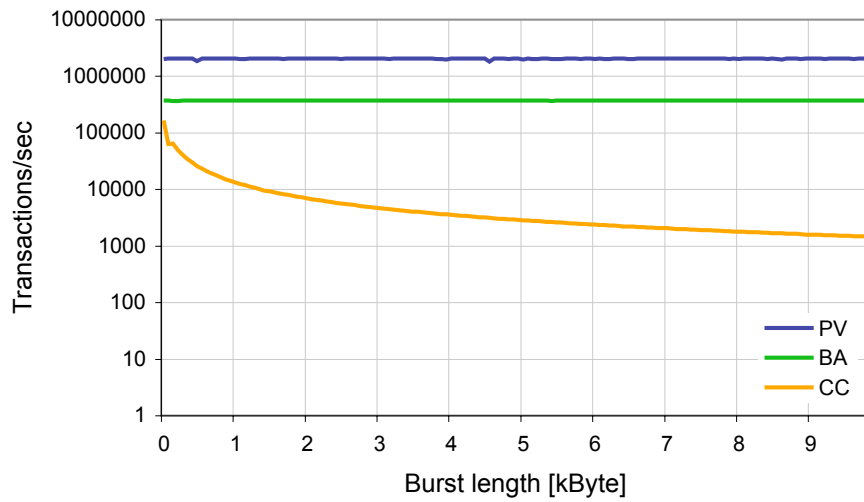


Figure 4.27: PV - BA - CC performance comparison of GREENBUS point-to-point writes

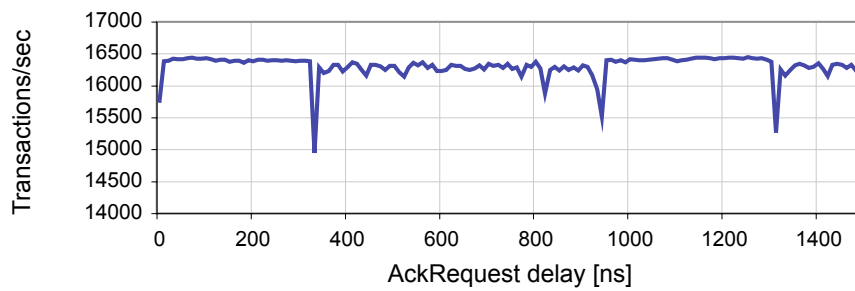


Figure 4.28: Impact of request acknowledge delay on GREENBUS performance

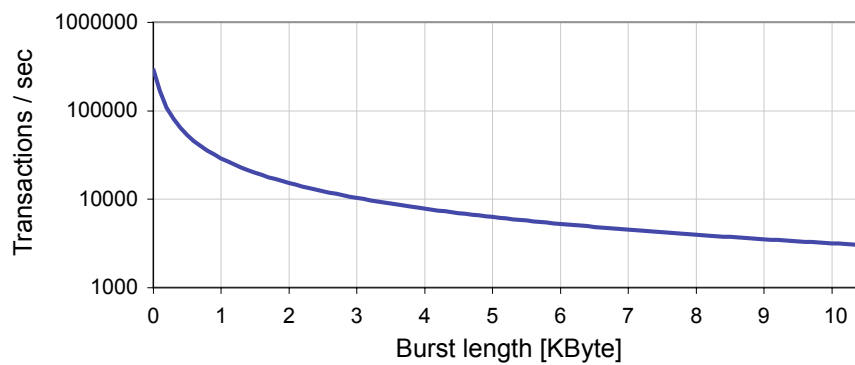


Figure 4.29: Example of BA ⇔ CC mixed-mode communication: TAC to OCP-tl1 writes

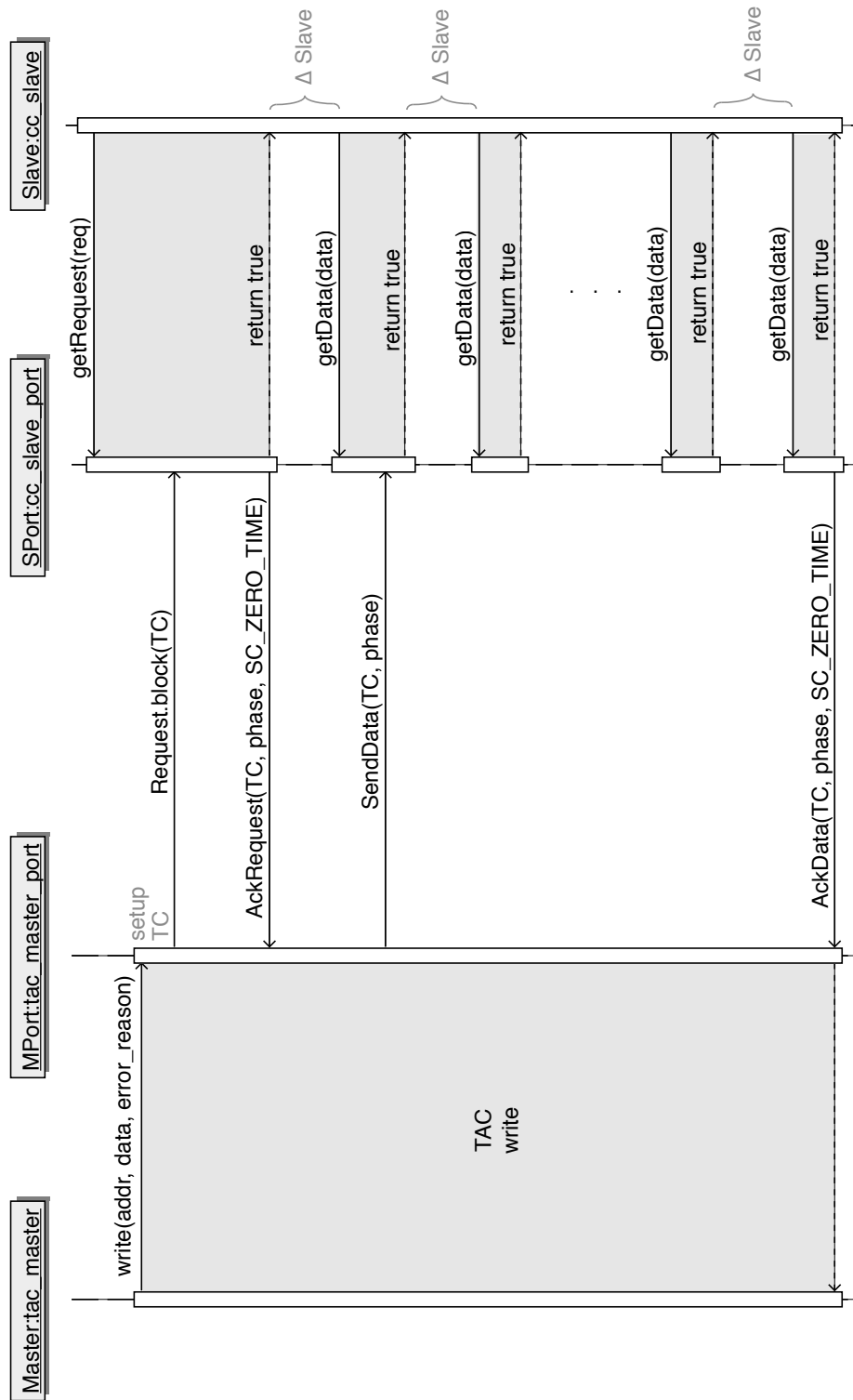


Figure 4.30: BA communication between a CCATB accurate and a CC accurate user API

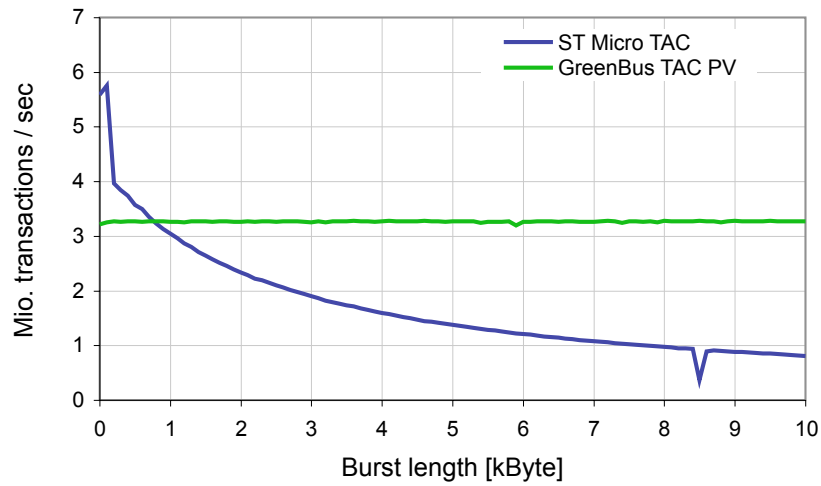
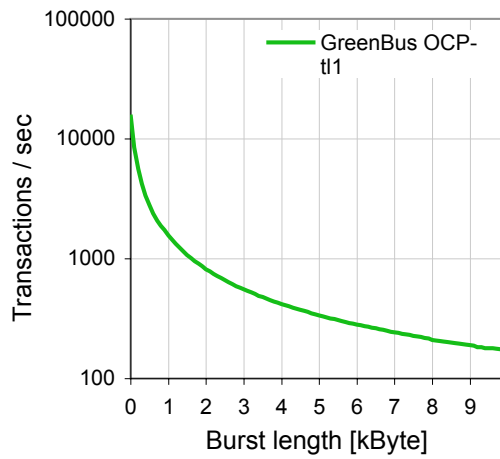
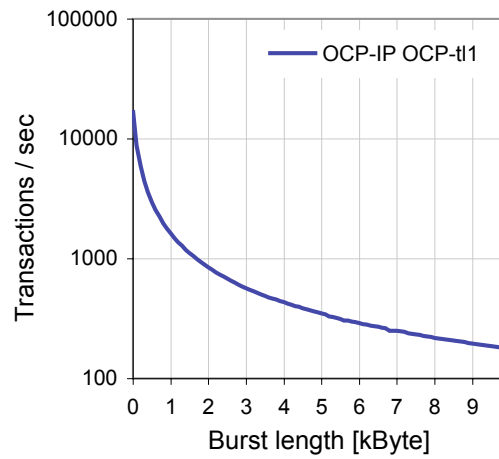


Figure 4.31: PV performance: comparison of TAC transactions over GREENBUS with TAC transactions over ST's TAC fabric



(a) OCP-tl1 transactions over GREENBUS



(b) OCP-tl1 transactions over OCP channel

Figure 4.32: OCP-tl1 simulation performance of GREENBUS compared to the original OCP-IP channel

difference whether the `AckRequest` atom is scheduled for delivery in 10 ns, 100 ns, or 1000 ns. This confirms the TAQ approach to be suitable for the typical communication scenarios in TLM models.

Finally, figure 4.29 shows the simulation performance of a TAC master talking to an OCP-tl1 slave. This system has been built with the components from the previous experiments: I took the TAC master and simply connected its user API directly to the OCP-tl1 user API of the OCP-tl1 slave.

This demonstrates the interoperability capabilities of GreenBus. The simulation performance is better than in the OCP-tl1 \leftrightarrow OCP-tl1 experiment (CC curve in fig. 4.27), which results from the fact that the TAC master user API sends just one data atom for the whole write burst. The cycle-count accurate OCP data handshake sequence is exercised between the OCP-tl1 slave user API and the slave only. Figure 4.30 shows a message sequence chart that depicts this approach. To simplify the diagram and point out the relevant timing points only, for the slave instead of the complex OCP-tl1 interface a more general CC accurate interface is shown.

4.10.2.2 Comparison to related frameworks

ST Microelectronics TAC

Figure 4.31 compares the performance of TAC communication over GREENBUS with the original TAC fabric from ST Microelectronics.

The experimental results highlight the advantage of using pass-by-pointer in GREENBUS: the simulation performance remains constant, independent of the burst length. ST's TAC fabric, which uses pass-by-value, however, provides better performance for short writes. The reason is that my TAC user APIs for GREENBUS suffer from the overhead that results from setting and interpreting the TC quarks for each transaction.

OCP-tl1

This experiment compares the performance of OCP-tl1 communication over GREENBUS with the performance of the OCP-tl1 channels from OCP-IP [95]. The measurements show GREENBUS to almost reach the same speed than the original OCP-tl1 channel. Note that for both measurements the same PEs were used – only the OCP-tl1 channel has been replaced with two GREENBUS ports that were directly bound together, namely the OCP-tl1 master port and the OCP-tl1 slave port.

All experiments were conducted on an Intel Core 2 CPU@2.4GHz machine with 2 GB RAM running Gentoo Linux 2.6.

4.11 Summary and outlook

This chapter presented the TAQ approach and a SystemC fabric to support it – GREENBUS. From the survey of existing TLM frameworks (chapter 2) three main requirements for efficient TLM communication modeling with SystemC were identified:

- R1 Standardization proposal:** what needs to be standardized and why;
- R2 Ease of use:** the approach should be intuitive, easy to use, interoperable, and error preventive;
- R3 Performance:** highest simulation speed should be attained.

For R1, I suggested both data structures and interface APIs which need to be standardized in order to make interoperation a reality. A systematic approach to abstract communication modeling and terms for these aspects were introduced (atoms, quarks, transaction container, payload events). Finally, a generic protocol was presented that enables interoperation among heterogeneous TLM user APIs. Different extension mechanisms have been developed and compared for this approach.

Table 4.9: Implementation of the user-view requirements from table 4.1 in GREENBUS

Req.	Description	Proposed technique	GreenBus implementation	Section
R2.1	Support different user APIs	Layered approach	Basic ports, generic ports, user APIs	4.3
R2.2	Generic protocol for data access and transport	Transaction container and generic protocol	Generic ports	4.5, 4.7
R2.3	Mixed-mode modeling, avoid adapters	Decouple transport mechanism and data representation	Basic ports, generic ports	4.3, 4.6
R2.4	Enable both untimed performance and timed architecture simulation with one model	Blocking and nonblocking low-level transport	Basic ports	4.6
R2.5	Automatic memory management	Transaction pooling, shared pointer	Initiator port	4.6.3.1, 4.10.1
R2.6	Clearness about transaction data validity and life span	Atom life span, generic protocol	Generic ports	4.5, 4.7
R2.7	Clearness about abstraction	TAQ abstraction formalism	Basic ports	4.4.1, 4.6.1
R2.8	Clearness about communication timing	Nonblocking transport interface	Payload event queue	4.6.2

My GREENBUS reference implementation for SystemC implements all these techniques and provides both good performance (R3) and an easy-to-use interface (R2). The experiments and comparison to related frameworks show that GREENBUS performance does not suffer from the layered approach. Table 4.9 summarizes the GREENBUS features in relation to the R2 requirements from the beginning of this chapter.

The key advantages of GREENBUS are:

1. clear distinction of standards from user code;
2. user and low level API are separated, and the low level API follows efficient user level convenience functions;
3. formalism for abstraction levels;
4. a simple ‘bus accurate’ level router can be used efficiently for communication architecture exploration. This presents the possibility of automatically generating virtual system prototypes from a description of the architecture features, using heterogeneous IP with different interfaces and abstractions.

The latter will be explored in the following chapters, which also will address the remaining R2 requirements (R2.9 to R2.13).

Part II

Architecture Exploration and Analysis

5 Communication Architecture Exploration

Contents

5.1	A router architecture for GreenBus	81
5.2	Scheduling and arbitration	84
5.3	Protocol simulation	87
5.4	Experiments	89
5.5	Discussion of GreenBus CAFM accuracy	98
5.6	Summary	100

5.1 A router architecture for GreenBus

While the generic protocol in combination with user APIs enables interoperability of PEs with heterogeneous interfaces, the generic router is that part of GREENBUS that is the principle part of any CAFM, the router and arbitration mechanism itself. Again, the approach I have taken keeps as much of the fabric re-usable as possible. The actual communication architecture simulation subsystem splits up into three pieces: the generic router, a bus protocol class, and a scheduler class. This architecture is shown in figure 5.1. The three components are interconnected by port-to-interface binding. The router is the generic part, that can be reused without change for any bus. In contrast the bus protocol class contains all the bus specific information. So the router in connection with an PLB bus protocol class forms an PLB bus functional model. In connection with a PCIe bus protocol class it forms a PCIe bus functional model. This decoupling of routing, which is common to all buses and bus behavior, which is very specific, is possible with the help of the previously described atom concept.

From my review of buses, the most important requirements for bus functional simulation are:

1. Support multiple simultaneous, outstanding and active transactions;
2. Support and profit from transaction's phase structures;
3. Support fixed and dynamic delays;
4. Events must mark rising signal edges to enable wrapping onto RTL;
5. Support clocked and combinatorial arbitration.

The router conforms to the GREENBUS architecture, implementing the two transport methods of its target port, blocking and nonblocking, at the BA level of abstraction. The blocking method (`b_transact(TC)`) takes a TC and transfers it as a whole to the targeted slave. This method is used for PV transactions. The nonblocking interface method puts atoms into the router (`notify(ATOM)`) and the protocol class will forward the atom to the targeted slave either immediately or delayed, in accordance with the implemented protocol and arbitration scheme. On their way back from the slave to the master, atoms again are passed through the protocol class.

5.1.1 Address map

The router uses the special multi-ports from section 4.6.3.4 to which an arbitrary number of masters and slaves can be connected. Masters are connected to the `GenericRouterTargetPort` and slaves to the `GenericRouterInitiatorPort` respectively. Before simulation starts an address map is generated

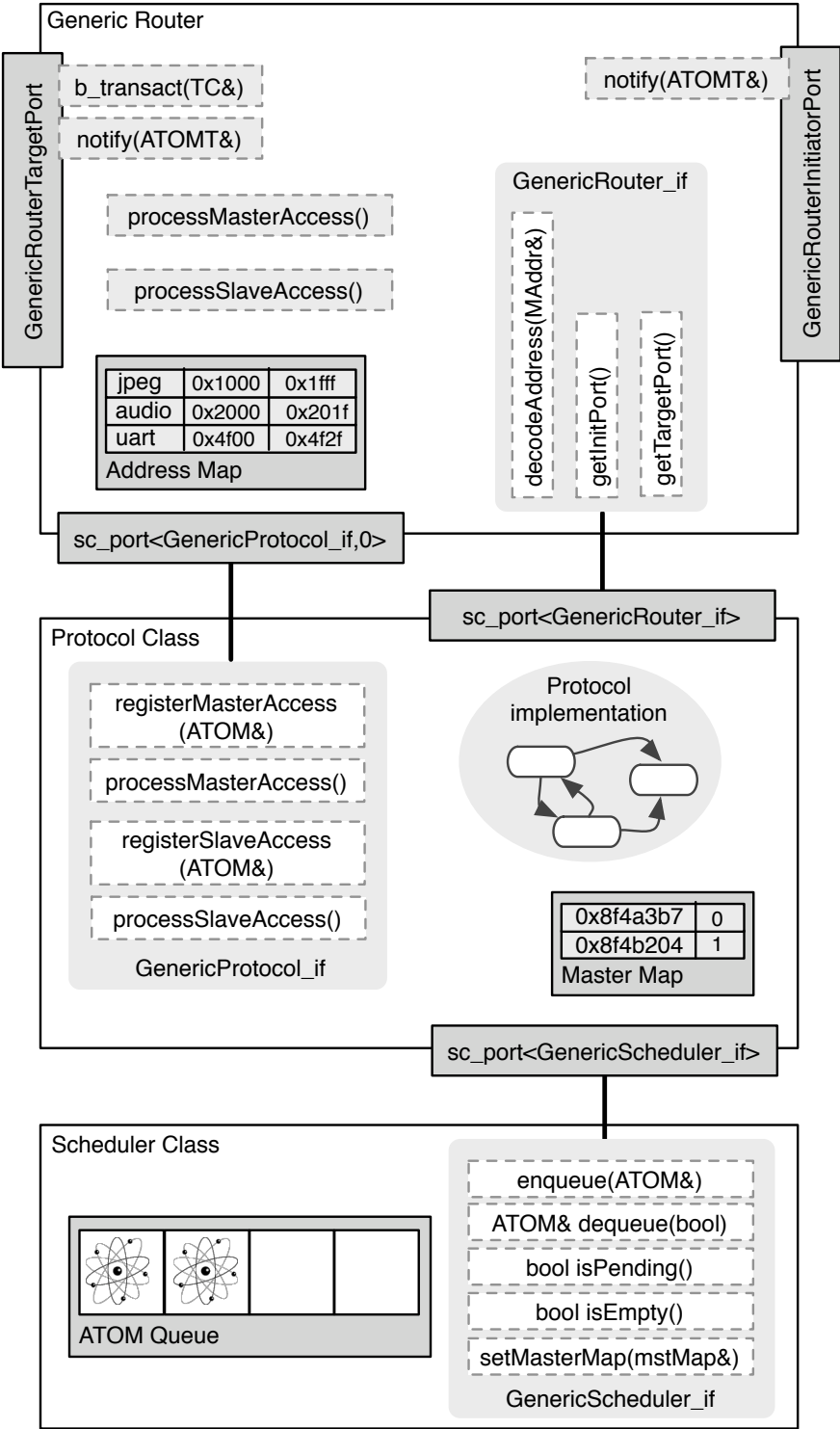


Figure 5.1: Bus functional simulation subsystem

for the connected slaves. For each slave this map contains its address range and given an address, it can decode the port index number of the `GenericRouterInitiatorPort` to which the targeted slave is connected.

With the GREENBUS reference implementation the following address maps are provided:

- *simple address map*: it supports one single contiguous address range per slave using the `base_addr` and `high_addr` target port parameters (cp. 4.6.3.2);
- *PCIe address map*: additionally enables ID-based routing such as used in PCI Express interconnects (see 5.4.2).

5.1.2 PV bypass mode

If a master uses the blocking `b_transact`, the router will do a `decode(ah->getMAddr())` call to the address map and then will call the blocking `b_transact` of the targeted slave. The protocol class is bypassed for PV transactions, which makes them very fast as the only simulation overhead generated by the router is for address decoding. The only place where a delay can be applied is in the slave. Figure 5.2 illustrates this behavior.

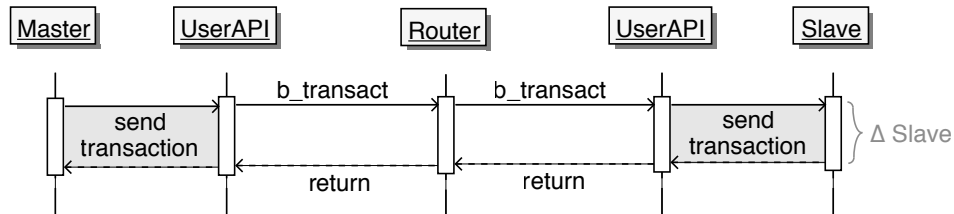


Figure 5.2: Transaction between a PV master and a PV slave over the generic router

5.1.3 BA/CC mode

If the master uses the nonblocking transport, the router's main task is to receive atoms from master ports and to deliver them to the targeted slave ports. Thereby the router must apply all the delays introduced in section 4.4.2.2. This has to be done in a generic manner, so that the router can be used for every conceivable bus. To this end, every time an atom is received from a master or a slave, the router does a callback into the bus protocol class, which is responsible for calculating and applying the delay.

Bus protocol classes must implement the `GenericProtocol_if` in order to support this mechanism. For atoms that arrive at the `GenericRouterTargetPort` of the router it calls the protocol's `registerMasterAccess` method. For atoms arriving at the `GenericRouterInitiatorPort` the `registerSlaveAccess` method is called. The router then suspends and waits for one of the following activities:

- A new atom arrives at one of the multi ports;
- The protocol class triggers `processMasterAccess` or `processSlaveAccess`.

The latter are `SC_METHODS` that can be made sensitive to protocol class internal events. For each received atom, the protocol class calculates its delay and schedules its processing for the future. This is indicated by an event, which executes the router's `processMasterAccess` (for master atoms) or `processSlaveAccess` method. In the reference implementation, these methods do nothing but calling the protocol's methods of the same name. However, they are there to enable user extensions to the router's atom processing functionality. After the protocol's `processMasterAccess` or

`processSlaveAccess` method has been called, it's up to the protocol class to handle the pending atom(s) and eventually pass them to the targeted receiver. The protocol class makes use of the `decodeAddress`, `getInitPort`, and `getTargetPort` methods with which all essential router elements (address map, target port, and initiator port) can be accessed through the `GenericRouter_if`.

Figure 5.3 illustrates BA \Leftrightarrow BA transactions over this architecture. Mixed-mode transactions between PV and BA PEs is inherently supported because the mapping of abstract transaction onto less abstract transactions and vice versa is done in the user APIs, not in the router. Figure 5.4 exemplifies this for PV \Leftrightarrow CC transactions.

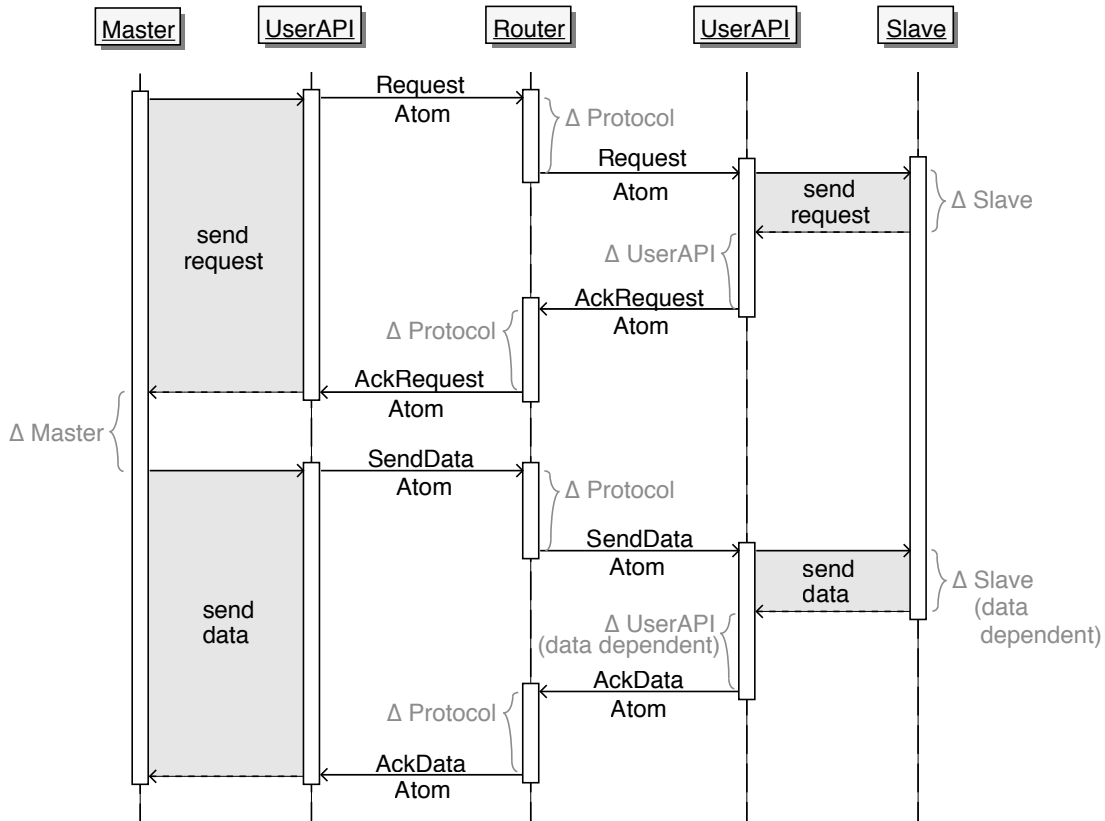


Figure 5.3: Transaction between a CCATB accurate master and a CCATB accurate slave over the generic router

5.2 Scheduling and arbitration

When multiple masters are connected to the router, there can be simultaneously incoming request atoms competing for access to the bus. To solve such conflicts, a scheduler is used that is attached to the protocol class via the `GenericScheduler_if` interface. For each request atom, the protocol class does an `enqueue(ATOM)` call to pass the atom to the scheduler, which inserts it into a request queue. Different arbitration policies can be implemented with this approach by simply changing the sort key for this queue. For example, a fixed priority scheduler would use the connection order of the masters as sort criterion, whereas a dynamic priority scheduler would consider the value of a priority quark in the TC.

The schedulers provided with the reference implementation of GREENBUS use the C++ standard template library (STL) `multiset` as atom queue:

```
typedef std::multiset<ATOM, atomCmp> atomSet;
```

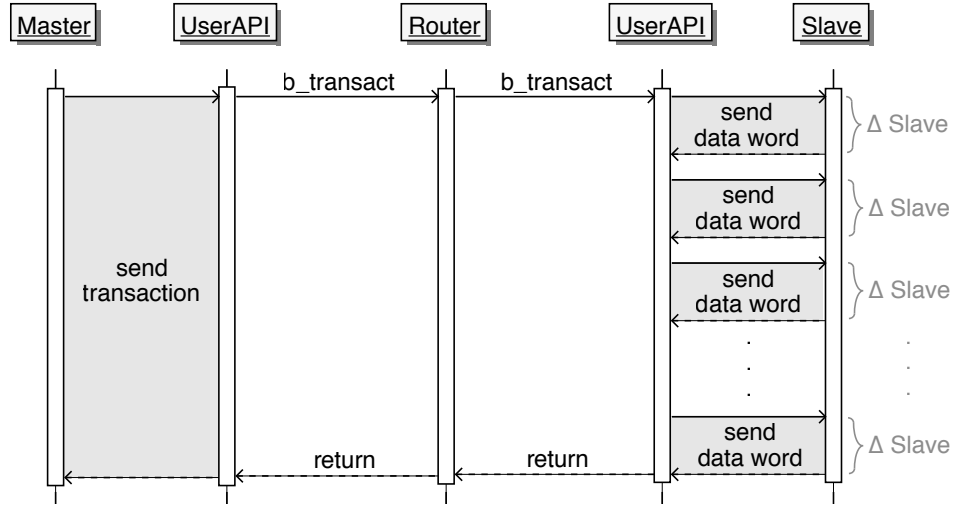


Figure 5.4: Mixed-mode transaction between a PV master and a CC accurate slave over the generic router

Listing 5.1: Comparison class implementation for a fixed priority scheduler

```

1  template <class T>
2  class fixedPrioritySchedulerCmp {
3  public:
4      fixedPrioritySchedulerCmp(std::map<gs_uint32, gs_uint32> *mstMap)
5          : m_pMstMap(mstMap)
6      {}
7
8      bool operator() (const T& x, const T& y) {
9          GenericRouterAccess& t1 = x.first->getRouterAccess();
10         GenericRouterAccess& t2 = y.first->getRouterAccess();
11
12         // compare master connection order
13         return (*m_pMstMap)[t1.getMID()] < (*m_pMstMap)[t2.getMID()];
14     }
15
16 protected:
17     std::map<gs_uint32, gs_uint32> * m_pMstMap;
18 };
  
```

The sort key is implemented by the `atomCmp` class that is passed to the `multiset` as template parameter. Thus, it can be easily exchanged while the code for the scheduler itself is reused. Listing 5.1 shows the `atomCmp` class for a fixed priority scheduler¹. A ‘master map’ is used that returns the port index for a given master ID. This map is created by the protocol class before start of simulation. When a new atom is inserted into the scheduler queue, the compare function reads the master ID and determines the position in the queue by evaluating the corresponding master port index.

In order to discover which request atom can be granted the protocol class calls `dequeue` on the scheduler. For cycled arbitration, this will be done with the rising edge of a simulated clock. This can be realized quite easily by scheduling an internal event for the duration of one clock cycle after the first request atom has been received. This technique is considerably faster than using an `sc_clock` (as e.g. can be seen in the OSSS framework [42] and in IBM’s CoreConnect channels [57]), because the rising edge event is only fired when arbitration actually takes place. Again, this advantage arises from the atom-based simulation approach.

Combinatorial arbitration, however, is another story. Many bus fabrics for SystemC do not support combinatorial arbitration at all (e.g., [25, 42, 57, 78, 116, 119]). One reason is that in a transaction level model, concurrent requests that are issued from different masters at the same time point may arrive at different delta cycles (cp. explanation 1). Often this is caused by `sc_signal` chains that occur in module hierarchies: a submodule forwards a signal to its parent module using an `sc_signal` and the parent module performs a bus access upon activity of this signal. The `sc_signal` delays the delivery of the value change by one delta cycle. Thus, the bus access will take place one delta cycle later than the bus access of the competing modules.

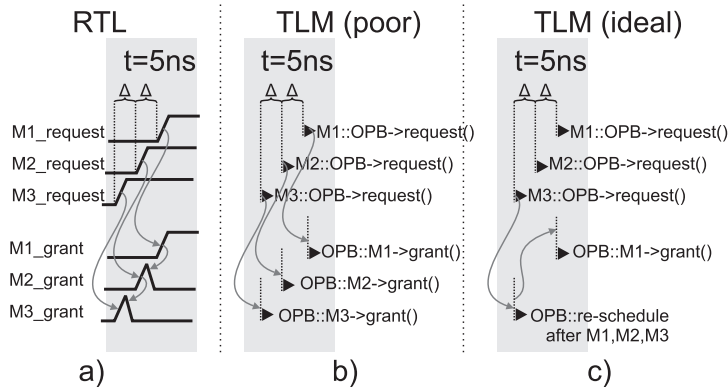


Figure 5.5: Combinatorial arbitration in RTL and TLM models

Figure 5.5 exemplifies this behavior with three masters trying to access a combinatorial arbitrated OPB concurrently: the requests are issued at the same time point but in different delta cycles.

In a RTL simulation (fig. 5.5a) this is not a problem: the grant signals get re-evaluated with every change of one of the request signals and thus indeed produce false intermediate results, but also the combinatorial logic of the three master models will be re-calculated in each delta cycle, such that eventually the simulation produces a correct result (at the end of the simulation time point). However, as is known this slows down simulation performance.

In a TLM model the arbiter therefore should generate only exactly one grant event per simulation time point. This would avoid simulation overhead due to re-arbitration. This ideal behavior is shown in figure 5.5c. TLM fabrics that do not care about delta cycle skew, however, will produce a behavior similar to figure 5.5b, which – if not properly considered by the model developer – may lead to unexpected model behavior and false simulation results.

¹The fixed priority scheduler and a dynamic priority scheduler are part of the GREENBUS reference implementation.

Implementation of the behavior from fig. 5.5c is everything but trivial. The root of the problem is that the bus functional model cannot know if a request is the last request at the current simulation time point or if further requests will arrive. Hence an algorithm is required that assures to not announce the arbitration decision before the very last request has arrived. Several approaches have been proposed to solve this problem but they all entail drawbacks either for the designer (higher modeling effort) or according simulation performance. [47] provides a survey of the existing techniques and in this paper we also propose a new approach based on *synchronization layers*. In comparison with the already existing solutions this novel approach shows excellent performance while avoiding their disadvantages.

5.3 Protocol simulation

To the generic router different protocol classes can be attached to simulate different protocols. The framework allows to do such replacements without recompilation, so that the exploration process can be automated, e.g. by using a shell script.

Each protocol class contains a state machine that reacts on atom arrivals and processes them in accordance with the simulated communication protocol. It thereby makes use of the attached scheduler class. Figure 5.6 shows the state machine for a simple bus protocol that allows for one read or write transaction at a time.

The state machine starts with the `wait for activity` state. Upon arrival of an atom from a master, the protocol's `registerMasterAccess` method is called. It checks the atom phase and if it is a request atom it calls the scheduler's `enqueue` method and schedules the start of the `processMasterAccess` method for the next clock cycle:

```
processMasterAccessEvent.notify(m_clkPeriod);
```

The protocol class now again waits for activity. Further master requests may be enqueued until the `processMasterAccessEvent` fires. Then the protocol's `processMasterAccess` method is invoked, which gets the top-most pending request from the scheduler and passes it to the targeted slave. When the slave acknowledges the request, `registerSlaveAccess` is called and the `AckRequest` is passed to the master. The bus state changes from 'idle' to 'busy' and now it is ready to transport data atoms from master to slave (for a write) or response atoms from slave to master (for a read). In this 'data phase' the protocol's sole job is to hand over the atoms among master and slave, thus representing a 'virtual' point-to-point link. The protocol class monitors the traffic and eventually detects the finalize atom. The bus state then changes back to 'idle', thereby again notifying the `processMasterAccessEvent` to process the next request atom in the scheduler's queue, if any.

To summarize this, the call back `processMasterAccess` is used to determine $t_{deliver}$ from fig. 4.5 (page 43), while `registerMasterAccess` and the external bus specific scheduler class determine the arbitration scheme and t_{grant} . Similarly, $t_{terminate}$ is determined by the `registerSlaveAccess` and `processSlaveAccess` methods (the latter not being used in the above protocol example). It is important to notice that t_{accept} is not bus but slave dependent because the acception is triggered by the slave. The GREENBUS architecture enables the slave to immediately accept incoming atoms without calling `wait` but nonetheless applying a t_{accept} deferral using the payload event delayed notification mechanism (cp. 4.5.3).

All communication delays can be modeled either fixed or to be dynamically dependent on the transaction. For example, a priority quark can be used to influence the scheduler's arbitration decision. Similarly, t_{accept} typically will depend on the transaction burst length. Finally, configurable parameters (6.2) can be used in the protocol and scheduler classes to modify communication parameters and thus explore the effects of changes to critical architecture properties.

The protocol depicted by figure 5.6 actually is similar to the IBM CoreConnect OPB protocol with cycled arbitration. Only two callbacks into the protocol class and one into the scheduler class are made per request atom, and only one callback is required for all other atoms. No explicit `wait` calls

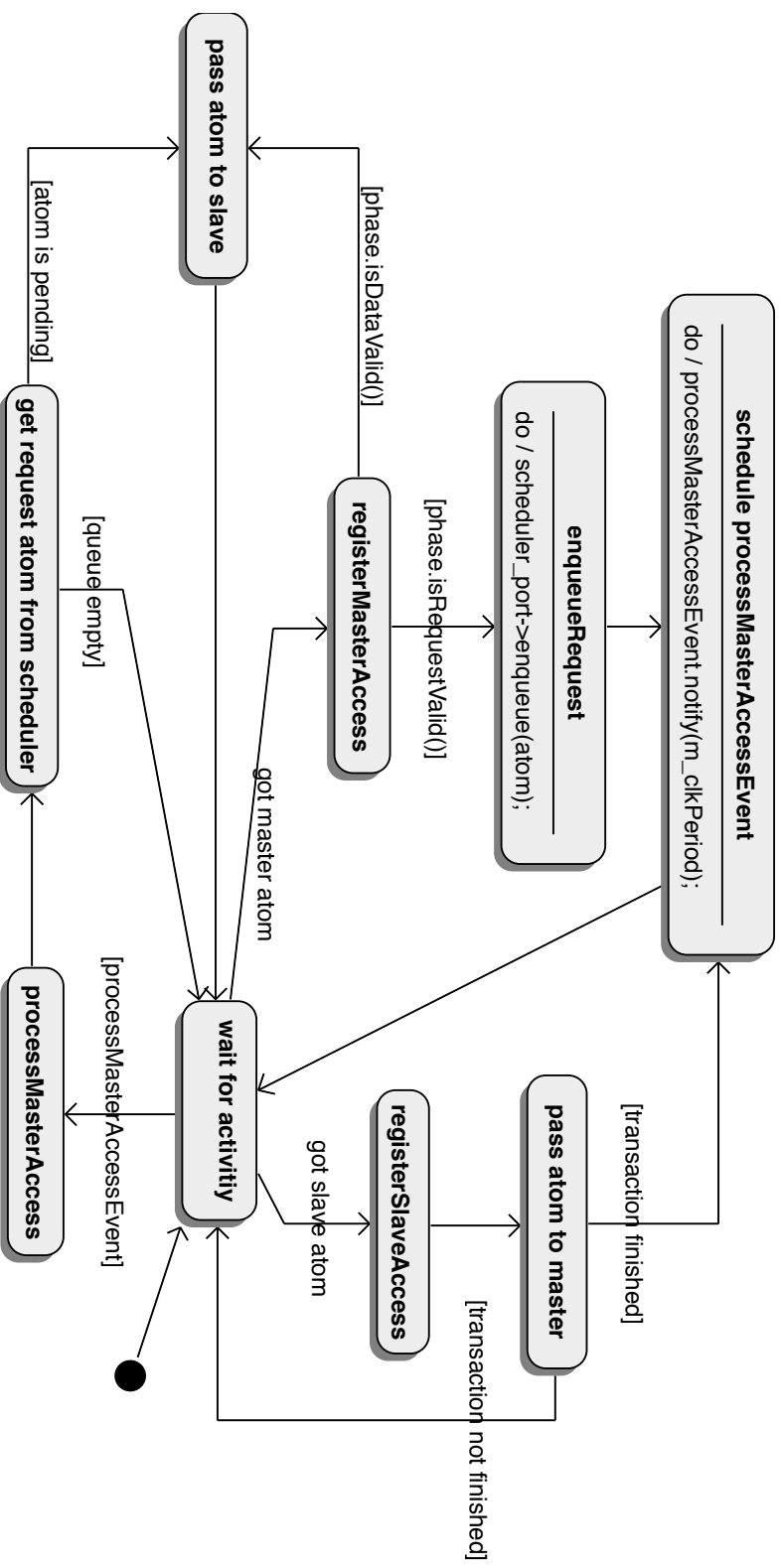


Figure 5.6: OPB protocol class state machine

are made at all. Simulation of an OPB locked back-to-back write burst transfer² using the SRMD design pattern from appendix B at BA abstraction therefore only requires

$$3 \text{ (Request)} + 1 \text{ (AckRequest)} + 1 \text{ (SendData)} + 1 \text{ (AckData)} = 6$$

SC_METHOD call backs and scheduling of one additional `sc_event`. The resulting simulation output is a bus accurate representation of the OPB communication behavior in terms of protocol timing and arbitration.

Switching from BA to CC transactions does not necessitate any changes to the bus protocol or scheduler classes. Once the request has been granted the rest of the transaction is under control of the master and slave user APIs.

In conclusion, this communication architecture simulation concept contributes to the requirements given on page 81 by making use of the atom concept (requirement 2), by handling multiple atoms at once (requirement 1), by supporting both fixed and dynamic delay action (requirement 3), by triggering events at the positive edge of a signal (requirement 4) and by supporting an efficient mechanism for combinatorial arbitration (requirement 5).

5.3.1 Mapping protocols onto atoms

The above presented mapping of the OPB protocol onto GREENBUS atoms is straightforward, as the OPB can only handle one transaction at the same time. More complex bus protocols, such as used by PLB and AXI, require a deeper understanding of their various (parallel) operations in order to derive an adequate abstraction of their RTL signal onto atoms. Our experiments with different buses have shown the following approach to deliver satisfying results:

- Consider the arbitration signals of the bus (i.e. those signals that are connected to the arbiter): which conditions must be met in order to successfully gain bus access? From these the extent of the ‘request phases’ supported by the protocol are derived.
- Consider the transfer qualifier signals of the bus (i.e. those signals that control the data flow): which conditions must be met in order to read/write data words? From these the extent of the ‘data phases’ supported by the protocol are derived.
- Reconsider the arbitration signals of the bus: which signals determine whether a transaction is terminated? From these the extent of the ‘acknowledge phases’ supported by the protocol are derived.

To verify the quality of a mapping, a waveform signal trace can be generated with GREENBUS for different test scenarios (regression test suites are provided for most buses). A comparison with the output of a cycle accurate model of the bus reveals timing errors. More sophisticated testing approaches may include formal verification techniques, but these are out of the scope of this thesis.

If a protocol specification is available as a formal definition (e.g., state machines), this might present the possibility to automatically translate it into a BA CAFM. However, for this to work additional annotations in the formal model would be required with which the adequate protocol phases can be identified. Again, this is out of the scope of this thesis.

5.4 Experiments

5.4.1 Processor Local Bus (PLB)

This chapter evaluates the capabilities of the bus functional simulation subsystem using the PLB as example. In comparison to the OPB, a protocol class for the PLB must support the following additional features:

²Compare [55, pg. 44].

1. *Dynamic priority based arbitration:* a master can set its bus access priority for each request individually.
2. *Overlapping of read and write transfers:* the PLB architecture has two distinct data buses that enable concurrent reads and writes.
3. *Address pipelining:* a master request can be granted as a ‘secondary request’ while the bus is still busy, thus avoiding the one-cycle arbitration delay for the secondary request.

The first requirement can be met by adding a new `mPrio` quark to the transaction container (see 4.8) and creating a new scheduler that can handle dynamic priorities based on this quark.

The second requirement affects the state machine implementation in the PLB protocol class. We can use the OPB state machine from figure 5.6 as a base and extend the control flow so that if there is a pending request atom in the scheduler queue and it can be processed in parallel to an already running transaction, this atom is granted immediately.

For the third requirement we introduce the three new phases that have already been shown on page 68: `SecondaryRequest`, `WrPrim`, and `RdPrim`. With these the protocol can announce secondary requests (that are on hold) to the slave as well as their promotion to primary. A PLB slave can use this information to prepare its reply to a request while the request is still on hold.

The PLB implementation encompasses 730 lines of code, whereas the OPB protocol class gets by with 300 lines of code. This highlights the complexity of the PLB but also shows the modeling efficiency of the GREENBUS approach. For example, only the header files³ of IBM’s PLB models for SystemC already have more than 500 lines, not counting the comments.

5.4.1.1 Simulation accuracy

To evaluate the quality of the PLB simulation with GREENBUS I have compared the simulation results with both IBM’s simulation framework for SystemC [57] and VHDL models from Xilinx [132]. The experimental set ups are shown in figure 5.7. A producer (master) and a consumer (slave) component have been modeled both in SystemC and in Verilog. The SystemC models do either PV, BA, or CC transactions and can be connected both to a GREENBUS PLB bus functional model or to IBM’s SystemC model of the PLB (the latter only supports CC accurate operation). The Verilog components are at the RTL level and use a pin and cycle accurate interface. They are fully synthesizable (using a hardware synthesis tool such as Xilinx XST [135]) and have been connected to the VHDL PLB model, which is part of Xilinx Embedded Development Kit (EDK [136]).

In the test set up, the two masters are continuously performing read and write bursts on the two slaves. The generated bus load can be controlled by the burst length and by wait times between the transactions. The test exhausts all the special PLB features such as overlapping read and write transactions, dynamic priority arbitration, as well as address pipelining. I have used two techniques to evaluate the simulation results:

- Compare the simulated execution time of the three test set ups after a pre-defined number of read and write transactions have been performed.
- Verify the simulation output of the GREENBUS based simulation by generating a waveform trace and comparing it with the RTL waveform output.

The comparison of the simulated execution time revealed some minor protocol timing bugs in the PLB implementation for GREENBUS which were quickly identified and fixed using the waveform trace. The simulation with GREENBUS then produced exactly the same waveforms as the RTL simulation (of course only in terms of the signal value changes that are considered by the atoms in the GREENBUS model), and the simulated model execution time was concordant with the results I got from the IBM

³The IBM PLB models for SystemC are distributed as a pre-compiled library.

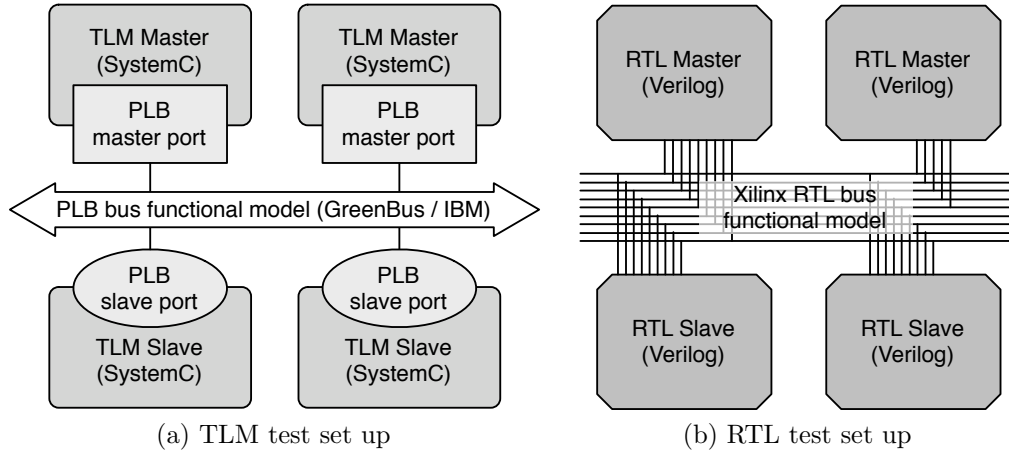


Figure 5.7: Evaluation of GREENBUS / PLB simulation quality

and the RTL models. Figure 5.8 shows a waveform output of the GREENBUS simulation, which can be created with GREENCONTROL (see 6.2). Both masters continuously perform write bursts. The two markers at 2080 ns and thereafter enframe one transaction of master M1 to slave S1. It can be seen that the write request at first is accepted as a secondary request and then promoted to primary after the running transaction of M2 to S2 has terminated.

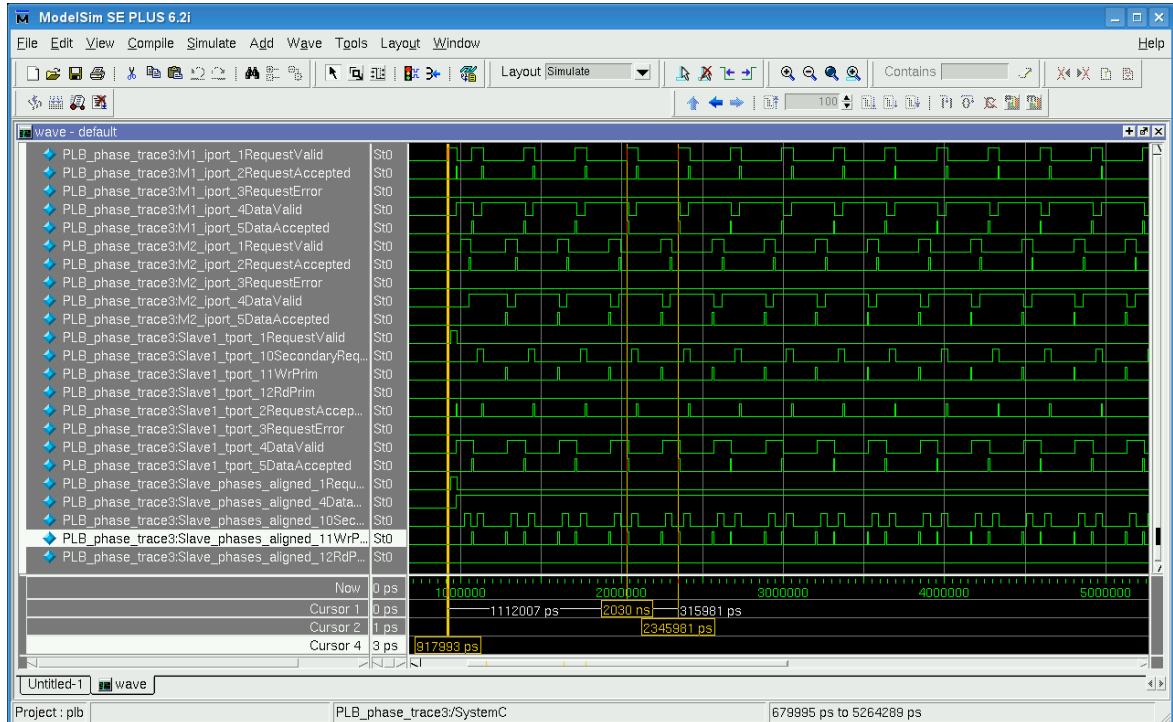


Figure 5.8: Waveform output of the GREENBUS PLB bus functional model at CC abstraction

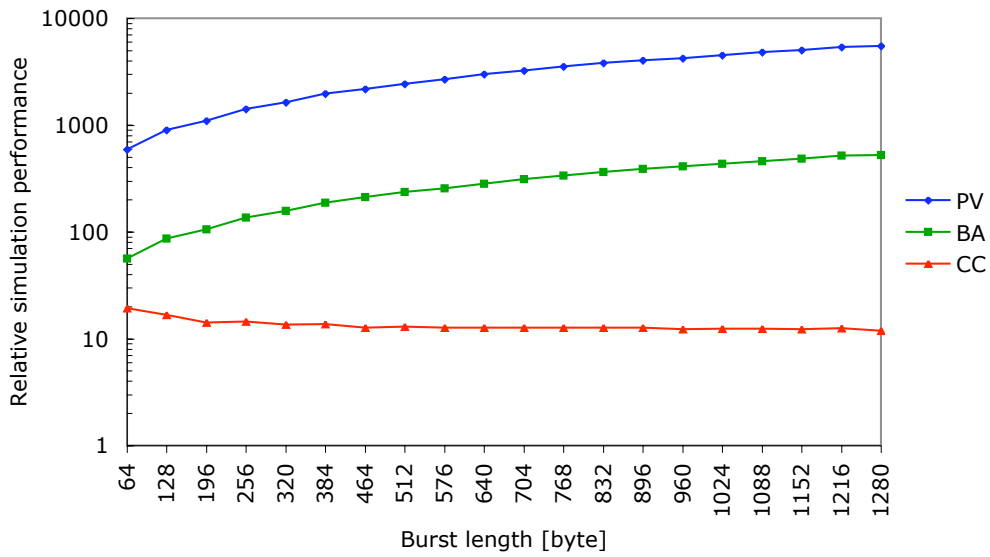


Figure 5.9: GREENBUS PLB simulation performance relative to Xilinx RTL bus functional model performance

5.4.1.2 Simulation performance

To examine the performance characteristics of communication architecture simulation with the generic router I compared the execution times for a simulation of 1 mio. PV, BA, and CC transactions respectively with the execution time of the RTL simulation, using the test bench from figure 5.7.

For all four tests I set the communication parameters to the same values (producing a bus load close to 100%) and any graphical and textual outputs have been suppressed. While the PV, BA, and CC transactions have been performed with SystemC 2.2 using the static cast variant of GREENBUS (4.8), for the simulation of the RTL model (VHDL model of PLB provided by Xilinx) the industry standard tool ModelSim 6.2 [87] was used.

Comparison of the different abstractions

It can be seen from figure 5.9 that the simulation speed increases with abstraction right similar as in the point-to-point experiment in chapter 4.10.2. In fact, there is nearly no difference; the overhead of the PLB simulation is insignificant. This confirms the TAQ approach to be an appropriate approach to bus functional simulation.

Comparison with the related work

To evaluate my PLB model in terms of the performance one can expect from dedicated bus functional models, I compared it to the work done by Schirner and Doemer from UC Irvine. In [112], the authors discuss hand-optimized TLM models for the AMBA bus family and present performance results for 512 byte transfers.

Table 5.1: GREENBUS PLB simulation performance of 512 byte writes relative to RTL, in comparison to the performance of AMBA models from Schirner et al. [112]

Model	PV	BA	CC	RTL
GREENBUS PLB	2442	233	13	1
UC Irvine AMBA	6802	78	-	1

Table 5.1 shows that the performance of GREENBUS and the performance provided by the AMBA models from Schirner et al. are in the same order of magnitude for the different abstractions. So we can say that concerning abstraction TAQ seems to work just fine. Note that the slower PV speed comes from the additional wrapping overhead in the user APIs which is not present in the dedicated AMBA models.

To evaluate GREENBUS performance in respect of absolute numbers consider figure 5.10. It shows the results I got from experiments with GREENBUS, IBM's PLB models, and the Xilinx VHDL model, respectively. According to my expectations, the performance of IBM's TLM model exceeds that of the VHDL simulation in the same order of magnitude than GREENBUS.

The measurements also show that GREENBUS running CC transactions scales very similar to IBM's model in terms of burst length. However, GREENBUS is about 4 times faster than the IBM model. As the source code of the latter is not available, the reasons are unclear.

The diagram also shows CCATB and PV performance I measured with GREENBUS (these abstractions are not supported by IBM's model). While the CCATB simulation reaches 250 thousand PLB transactions per second, at PV simulation speed increases to 2.7 mio. PLB transactions per second.

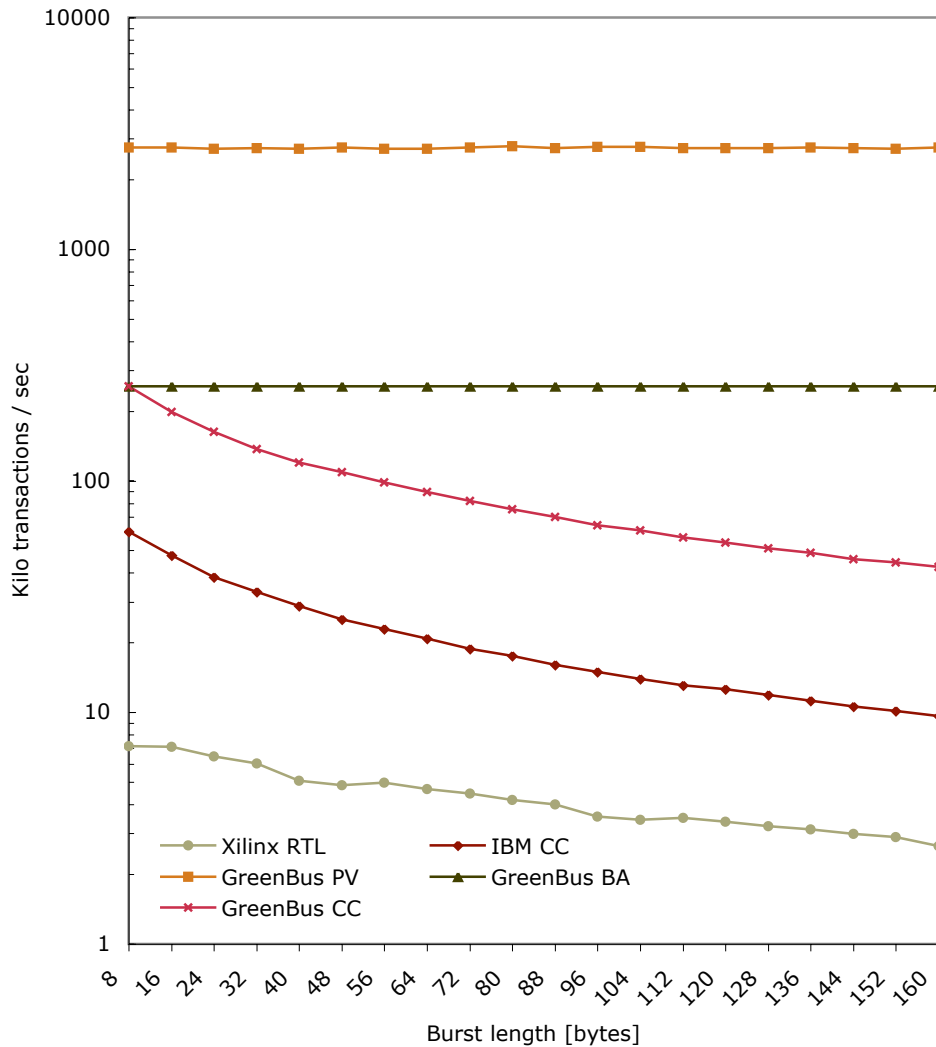


Figure 5.10: GREENBUS PLB simulation performance compared to IBM SystemC and Xilinx VHDL bus functional models

Impact of bus workload

A further experimental result is shown in figure 5.11. For this experiment 10 BA masters and 10 BA slaves have been connected to a GREENBUS router that simulates the PLB. The masters' task was to 'pollute' the bus by randomly sending 2 mio. write transactions of 1024 to the slaves. The experiment was carried out several times and for each collection of data the bus workload was increased by reducing the latency periods between transactions. The graph shows that GREENBUS simulation overhead isn't strongly affected by increased bus load. Surprisingly, for bus loads greater than 85% a slight performance increase of 10% can be seen. This results from less events scheduled at high bus contention, because there more often already is a pending request in the queue which can immediately be processed after the preceding transaction has been finalized. The number of payload events for the transactions themselves, however, does not change.

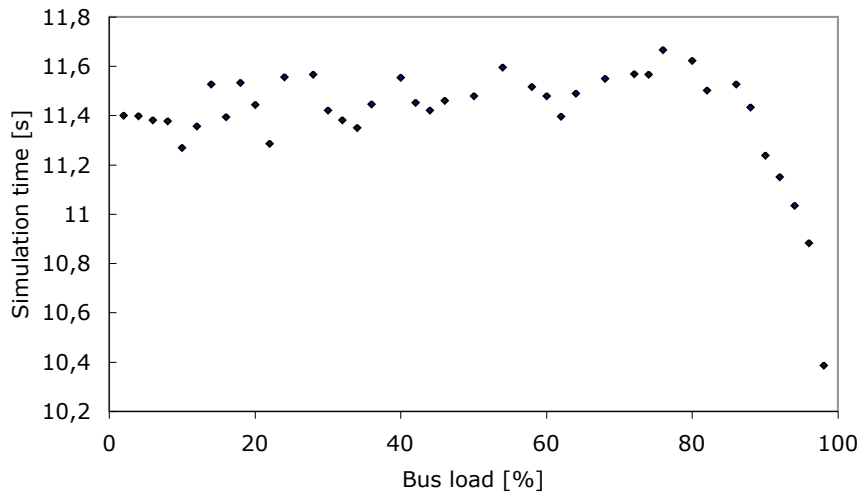


Figure 5.11: GREENBUS PLB simulation performance in dependence of bus load

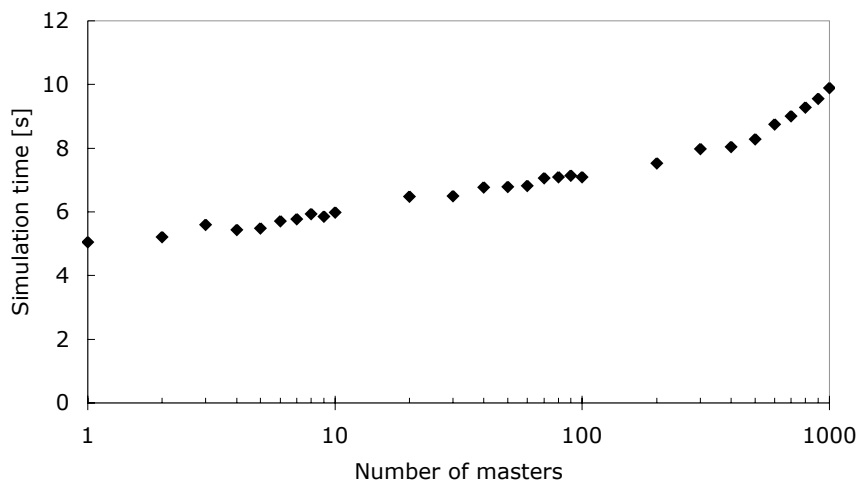


Figure 5.12: GREENBUS PLB simulation performance in dependence of PE number

Finally, figure 5.12 shows the PLB simulation performance in dependence of the number of masters accessing (and being connected to) the bus. For each measurement a total of 1 mio. transactions has been send, with each master doing $\frac{1,000,000}{\#mst}$ BA writes of 16 byte, interrupted by random delays to keep the overall bus load under 20%. Again, the results confirm the expectation: GREENBUS scales very well (note the logarithmic scale), as from the GREENBUS architecture point of view there is no reason why the number of components connected to the router should affect its simulation overhead. However, a slight performance drop down can be seen in the measurements, which results from increasing context switch overhead in the SystemC simulation kernel.

Note that all these experiments where constructed such that only the computation time spend in the GREENBUS CAFMs is measured, including router, bus protocol class, scheduler, all ports, and the user APIs. That is, the shown experimental results do not involve any (potential) simulation overhead that would be produced when using ‘real’ PEs. The experimental PEs do not consume any simulation time during transactions but the computation time needed for running through the protocol with the master and slave ports (i.e., interacting with the user APIs). The data transferred is random data and simply seeps away in the receiver, without doing any memory copies or the like.

5.4.2 PCI Express

In contrast to PLB, which has been designed for on-chip communication, PCI Express (PCIe) is a bus architecture for connecting multiple chips on a board. The PCIe standard [105] has been derived from PCI but in contrast, PCIe connections are always point-to-point. Figure 5.13 shows an example PCIe system.

In a research project funded by Intel Corp., Phoenix, we used GREENBUS as a base to develop a complete PCIe transaction-layer modeling framework for SystemC [114]. This work was particularly interesting in terms of two aspects:

- the PCIe standard defines an extremely versatile protocol and therefore its implementation for GREENBUS required extensive utilization of the extension mechanism (4.8);
- the ultimate goal was to take advantage of GREENBUS mixed-mode and interoperability features while at the same time highest simulation performance should be achieved.

The latter is particularly important for PV models: their intended use case is to enable device driver and firmware developers to efficiently develop, test and debug their code. Hence, simulation performance must be fast enough to boot an operation system (Windows, Linux) on the SystemC models.

The implementation of the whole PCIe modeling kit with GREENBUS took one person three months, including complete familiarization with both GREENBUS and the PCIe and PCI specification. The kit boasts a performance of *5.1 mio. transactions/sec* for PV memory reads/writes over the PCIeRouter, which is quite satisfying in consideration of the overhead due to mapping PCIe onto the GREENBUS generic protocol.

Figure 5.14 shows the PV performance of PCIe transactions in terms of GByte/sec via a PCIe switch. An average throughput of 7 GByte/s is reached on a 2.4 GHz Intel Core-2 linux PC. Note that only in the slave a memory copy takes place (copy-on-write, see 4.6.4). In conclusion, this project was a good test for the GREENBUS extension mechanism and the results show that compatibility of PCIe to the generic protocol can be achieved without significantly affecting simulation performance.

5.4.3 Network-on-Chip

In a research cooperation with GreenSocs and Texas Instruments the NoC simulation capabilities of GREENBUS have been examined and the outcome is a modeling that particularly addresses NoCs based on the Open Core Protocol. Two contributions have been made:

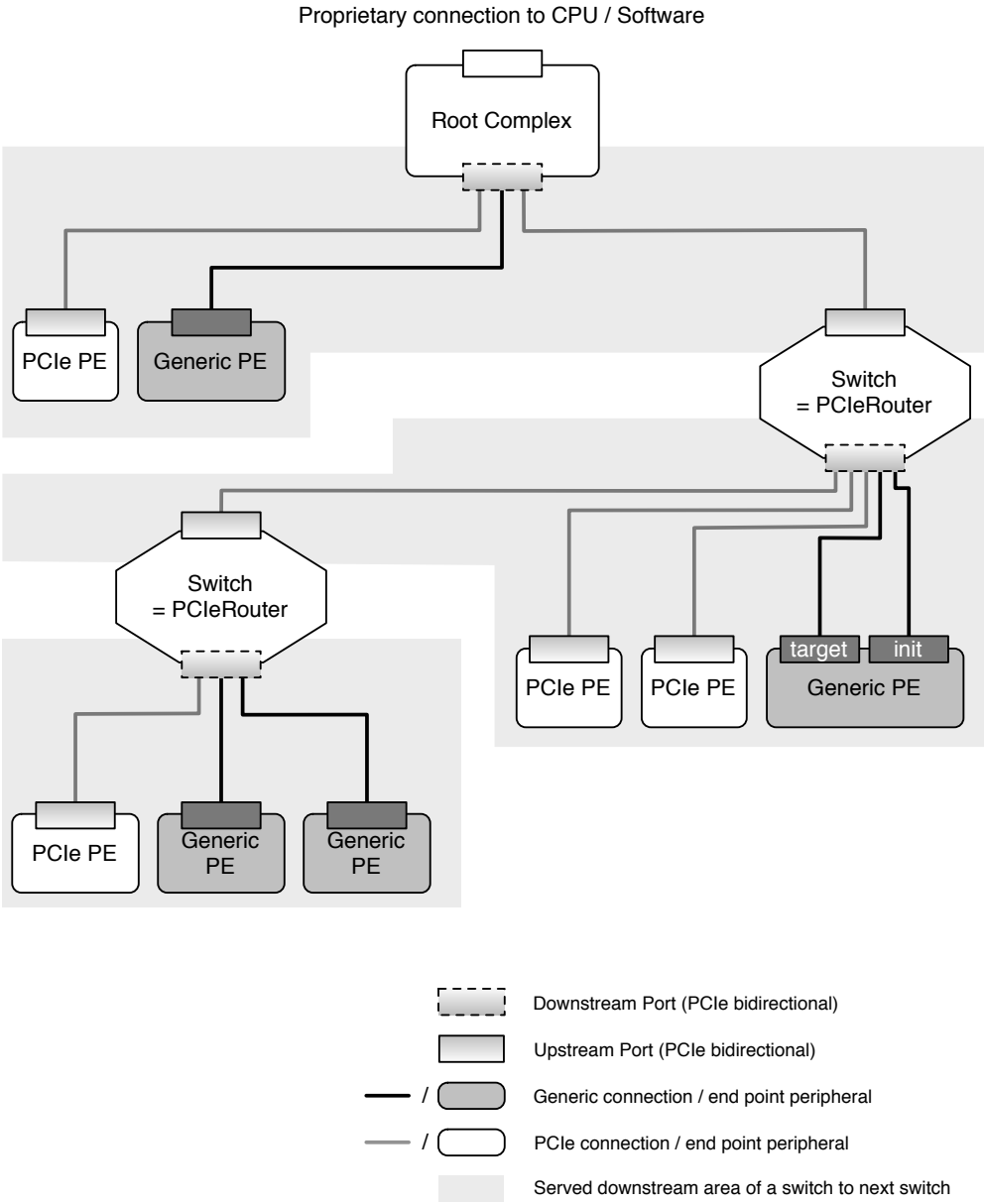


Figure 5.13: PCI Express example architecture

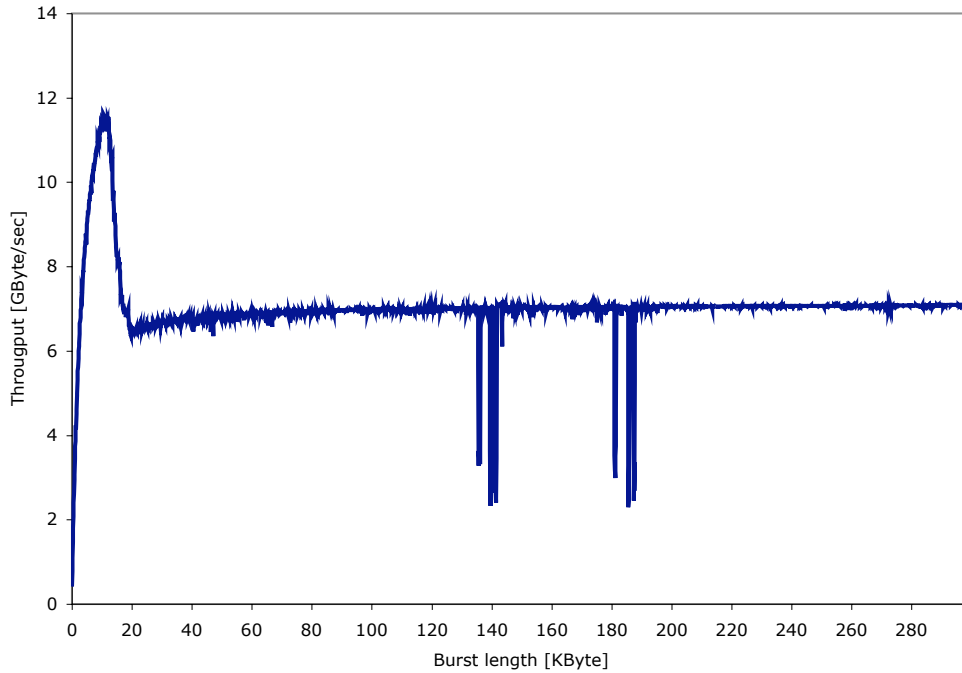


Figure 5.14: PV transaction performance (write bursts over a PCIe switch)

- A generic NoC switch based on the generic router;
- A NoC-enabled OCP transaction container.

An important result of this project is the observation that the data passed among PEs and NoC routers can be classified into four different categories, namely end to end invariant, point to point invariant, point to point changeable, and local data. For example, the command and the data of a transaction is end-to-end invariant, which means that it is not changed by the intermediate components while it ‘travels’ through the NoC. Other quarks, which are initially set by the master, however, may also be set by other communication components. For example, a bridge may overwrite the address quark due to address translation. When the transaction returns, it must recover the original value of this point-to-point invariant data. Our NoC transaction container implementation enables modeling of such behavior by providing special stacked quarks. With these, the original address quark value will be automatically recovered when the transaction returns to the initiator. This helps to reduce the housekeeping effort in routers and bridges. In [46] and [16] a detailed discussion of this approach is provided.

Figure 5.15 shows how the simulation performance in an OCP-based NoC modeled with our framework scales with the number of NoC switches on a transaction path. The diagram shows the results for test models where communication delays were applied at different locations in the system. The performance has been compared to a similar test fixture using the original OCP channels instead of GREENBUS. Unsurprisingly, the version with no delays is the fastest one, because the atoms can be immediately delivered by the payload event queues. If there are delays in both directions the speed up is reduced due to additional event scheduling. GREENBUS in all cases is faster and the source code for the NoC switches is simpler (less than half number of lines of code) than the code for the switches that use the original OCP channels, thanks to the ‘data life span aware’ quarks.

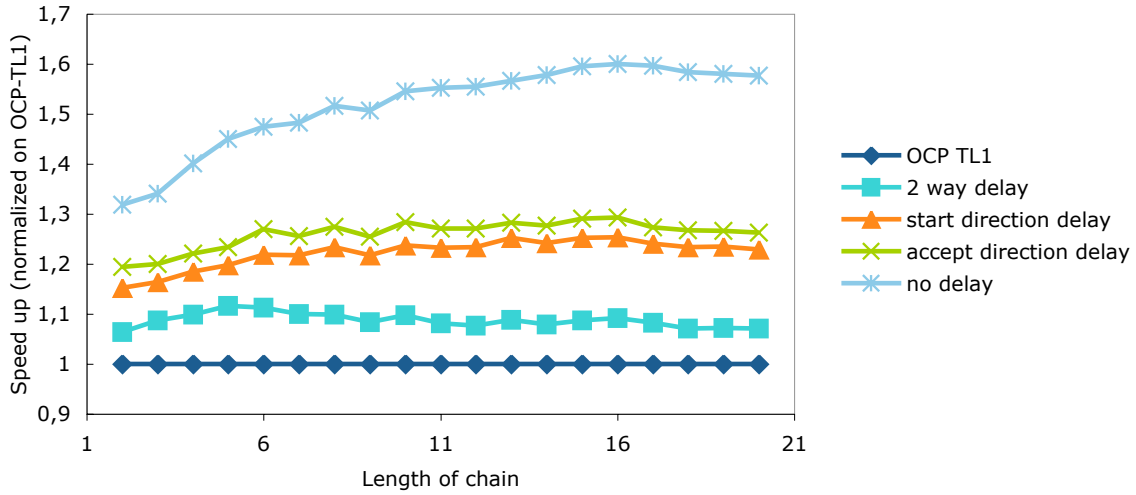


Figure 5.15: Write burst simulation performance of an OCP NoC model depending on number of hops (length of transaction path in terms of NoC switches)

5.5 Discussion of GreenBus CAFM accuracy

GREENBUS is intended to simulate buses (and similar communication links) at a ‘logical’ level. It does not consider electrical information. Instead, the simulation accuracy can be measured in terms of signals, clock cycles, and high and low levels. However, in contrast to RTL bus functional models, a GREENBUS CAFM does not have an exact internal representation of those signals. In their place, atoms and quarks are used as abstract representatives. The transaction container provides information about the current and all previous states of the communication, including information about the protocol, the originator, the addressee, the router(s), bridges, and so on.

With the nonblocking transport interface, CAFMs can deliver CCATB and CC accurate results. The difference between CCATB and CC is that at CCATB some blur is allowed concerning the timing fidelity of the data handshake. This enables the creation of fast abstract models: a BA PE can accept a complete data burst at once and if it can exactly estimate how long processing of this data will take, it can apply the accept delay for the data atom correctly (cp. 4.4.2.2). For example, the BA model of a memory controller can calculate the delay of a write burst like this:

$$t_{write} = t_{initial} + t_{data}$$

with

$$t_{initial} = t_{cycle}i, \quad t_{data} = t_{cycle} \frac{burstlength}{w}.$$

Here, it is assumed that after an initial number of wait states i for addressing the RAM, w bytes can be written per clock cycle t_{cycle} . For BA transactions, $t_{initial}$ is applied to the request atom and t_{data} to the data atom. For CC accuracy, the burst is divided into w data atoms and for each an individual t_{cycle} delay is applied. This is illustrated for a 9 byte write burst in figure 5.16.

Note that the figure only shows the delays applied by the memory controller. In a global scope, further delays will be added by the CAFM and by the initiator of the transaction (cp. figure 4.8). Only if the following conditions are met, a simulation with GREENBUS will be sufficient to produce CC accurate results at large:

1. the PEs produce precise accept delays for the data being transferred;
2. the communication architecture functional model does not allow transactions to be stalled or cut off during the data handshake phase (e.g., by a high priority device).

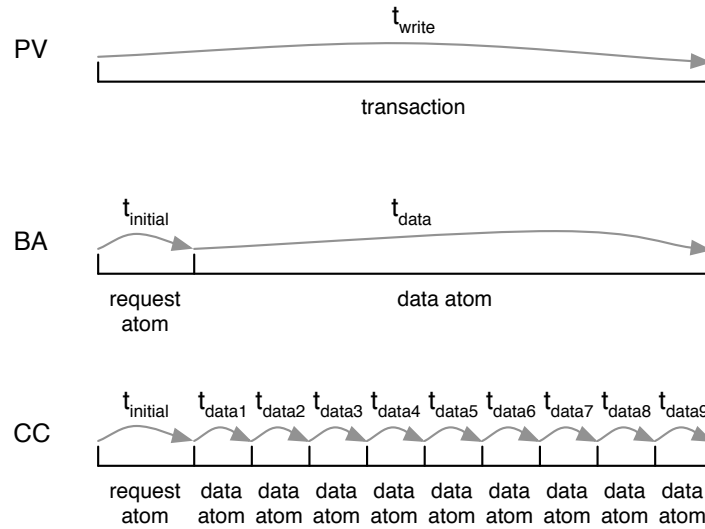


Figure 5.16: Communication delays applied by a memory controller PE at different abstractions

Condition 1 depends on the abstraction of the model, the modeled application, and the model of computation used. In Kahn process networks [65] and synchronous data flow models [84] for example, the total processing time for a transaction can be calculated by annotating each node in the data flow graph with a processing delay and then execute the model with the input data. However, execution of such models implies that the transaction data is fed into the computation block token by token. The PE computation abstraction therefore is lower than CCATB and thus such models rather should use a CC interface to the interconnect instead of internally chopping the BA transaction data atom into pieces.

In more abstract models it depends on the data dependency how good a delay estimation can be. For the memory controller from above the data accept delay can be calculated without even looking at the data. Only the burst length is important, so there is no data dependency at all. For a JPEG encoder, the delay cannot be exactly determined without actually performing the JPEG compression on the input data, so there is full data dependency. Other PEs such as network controllers may have partial data dependency.

Condition 2 is typically true for point-to-point links and simple buses. Their rather simple arbitration algorithms can be precisely dealt with in BA transactions. But if the protocol allows transactions to be stalled (e.g. AMBA split-transactions) or killed (e.g. CANbus high-priority messages [108]) during the data phase, only CC transactions can deliver cycle count accurate results. Figure 5.17 shows simulation of a stalled transaction at the different timing resolutions. At time point t_{start1} , a master issues a request and gets access to the bus. At time point t_{start2} , another master with higher priority accesses the bus. After the request is granted, the first transaction is paused and the data of the second transaction is transferred. Transaction 2 is finished at t_{term2} , so that transaction 1 resumes and terminates at t_{term1} .

With CC transactions, this behavior is simulated accurately. With BA transactions, the data phase of transaction 2 is not started before all data of transaction 1 has been sent, so the timing fidelity of the atom start and end time points is affected. But the total communication time remains correct. With PV transactions, the request delay is added to the delay estimation for transaction 2 and not only the intermediate simulation results but also the overall simulation result gets wrong.

In conclusion, the communication timing fidelity of a TLM model not only depends on the CAFM(s) but also on the abstraction of the PEs. The experiments in this chapter have shown that the TAQ approach allows to create CAFMs with unambiguous timing fidelity. The timing fidelity of PEs,

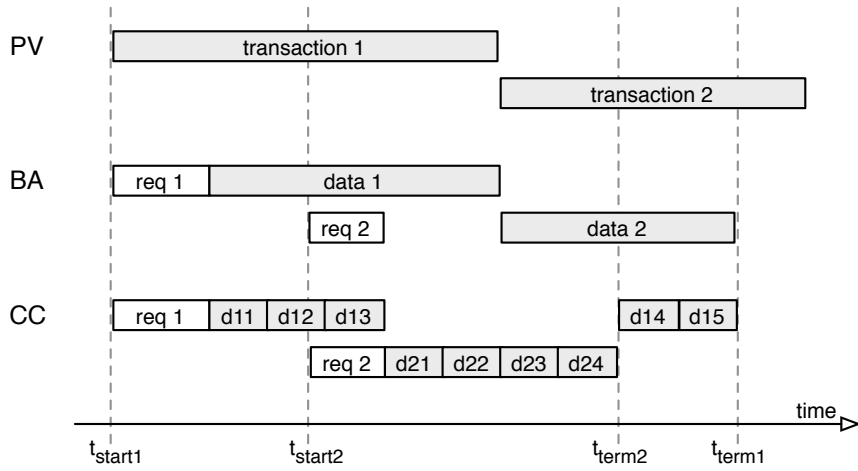


Figure 5.17: Simulation of a stalled transaction at different abstractions

however, depends on their underlying MoC and the individual coding style used by the engineer. TAQ and the generic protocol allow to model the communication interfaces of PEs with CCATB or CC accuracy, but further research is required to investigate how a predetermined timing fidelity for the computation part of PEs can be systematically achieved.

5.6 Summary

The router concept presented in this chapter aims at providing a generic building block for the construction of CAFMs with GREENBUS. It provides a plug-in interface for different protocol, address map, and scheduler implementations. Experiments with CoreConnect OPB and PLB, PCI Express, and the Open Core Protocol have shown the TAQ scheme to be a viable approach to abstract communication modeling. All examined protocols have been successfully mapped onto this scheme, although they differ quite significantly in structure and application. Also the simulation performance doesn't fall short of one's expectations. In fact, GREENBUS even outperforms its fully application-specific competitors, such as IBM's PLB models and OCP-IP's SystemC channels.

With this, requirements R2.9 to R2.11 (cp. table 4.1 on page 34) have been addressed, see table 5.2.

Table 5.2: Fulfillment of user-view requirements from table 4.1 by the generic router

Req.	Description	Proposed technique	GreenBus implementation	Section
R2.9	Provide a simple way of constructing interconnects	Router, bus protocol classes, schedulers, and user APIs	Layered approach, generic router	4.9, 5.1
R2.10	'Plug-In' interface for different CAFMs	Bus protocol and scheduler interface of the generic router	Port-to-interface binding	5.1, 5.2, 5.3
R2.11	Facilitate development of new user APIs and CAFMs	Combination of generic protocol and generic router	TAQ, design patterns, protocol state machines	4.7, 4.9, 5.1, 5.3, B

6 Performance Analysis and Visualization

Contents

6.1 Overview	101
6.2 GreenControl: a model instrumentation framework based on GreenBus	102
6.3 DUST: a SystemC-aware design analysis toolkit	106
6.4 Summary and outlook	110

6.1 Overview

For design space exploration in general and for communication architecture exploration in particular it is important to have appropriate support for:

1. *Testbench creation*: calibrate and configure PEs, create the interconnect, and generate stimuli data;
2. *Simulation control and debug*: start/stop and pause/resume simulation, set conditional breakpoints and assertions;
3. *Performance analysis*: during simulation collect performance data and send to database and/or analysis tool;
4. *Visualization*: show the performance trends of the design-under-test to the user with intuitive visualizations.

A well-known proposal for SystemC is the SystemC Verification Standard specification (SCV) [99]. Its implementation, the open-source SCV library, is an extension to SystemC focusing on test stimuli generation, model introspection, and transaction recording. While the abilities of SCV are quite impressive, its usability is rather not. Instrumenting a design for data introspection and transaction recording requires dozens lines of code and profound knowledge of SCV internals. Furthermore, SCV is slow: in our experiments with a video processor model, simulation performance with SCV transaction recording dropped under 3% of the original speed [71].

A second issue is debugging: SystemC does not have built-in debug support. Thus, analysis and debug functions well-known from RTL simulators such as pause/resume and conditional breakpoints are not supported. There do exist some mature development tools with SystemC-aware linter and debugging capabilities such as AccurateCTM [1], CoWare's Platform ArchitectTM [26], Celoxica's AgilityTMSystemC/RTL compiler [20], or Mentor's SystemC development environment VistaTM [86]. But all of these tools are either limited to a certain language subset [20] or application domain [26], and/or are bound to a specific version of the SystemC language, often in combination with a modified, proprietary simulation kernel [20, 26, 86].

In this chapter an approach to TLM-driven test, analysis, and debug is presented which is completely based on the idea of *configurable parameters*. Configurable parameters are variables that look and feel like normal data types but in addition provide interesting analysis and debug features. In the following the GREENCONTROL framework that reuses GREENBUS techniques to realize configurable parameters for SystemC will be discussed, and we will look at DUST a visual GREENBUS performance analysis tool written in Java.

6.2 GreenControl: a model instrumentation framework based on GreenBus

The GREENCONTROL¹ project aims to provide a library that can be used to connect user defined configuration mechanisms to tool configuration mechanisms, giving both flexibility. Figure 6.1 shows an IP block connected, via the GREENCONTROL library, to an ESL tool that is providing configuration from a database.

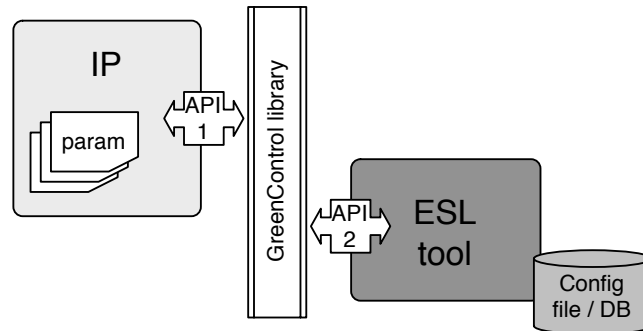


Figure 6.1: IP block, ESL tool, and GREENCONTROL library

The IP block has one API to the library. The library uses a different API to the ESL tool. This allows IP designers to choose the configuration mechanism that best suits their style of IP creation, while staying independent of tools. The GREENCONTROL framework is extendable with *plugins* which provide a service and add new functionality. One example is the configurable parameter plugin which we examine here. Other examples are²:

- *Analysis plugin*: collect statistical information of a design under test, such as bus workload, cache misses, etc., and forward it to a database or tool;
- *Register & memory plugin*: provide an API for the creation of memory regions and register banks that can be accessed both by the PEs and by analysis tools;
- *Debug plugin*: enable pause/resume simulation and conditional breakpoints (e.g., stop simulation when a parameter value changes or on violation of an assertion).

6.2.1 Requirements and related work

At the heart of the GREENCONTROL framework are configurable parameters. They are extended data types that can be used at any place in a SystemC design in replacement for a standard data type (including user-defined types). They aim to provide the following features:

1. *Model-wide parameter access* using hierarchical names;
2. *Initialization* using a configuration file or an external tool;
3. *Visibility*: find parameters using wildcards or regular expressions;
4. *Observation* of parameter value changes using callback function;
5. *Tool support* for debugging and visualization.

¹This research was funded by GreenSocs, Intel Corp., and Texas Instruments.

²At time of writing this document, the GREENCONTROL core library, the configurable parameter plugin, the debug plugin, and parts of the analysis plugin have been successfully implemented.

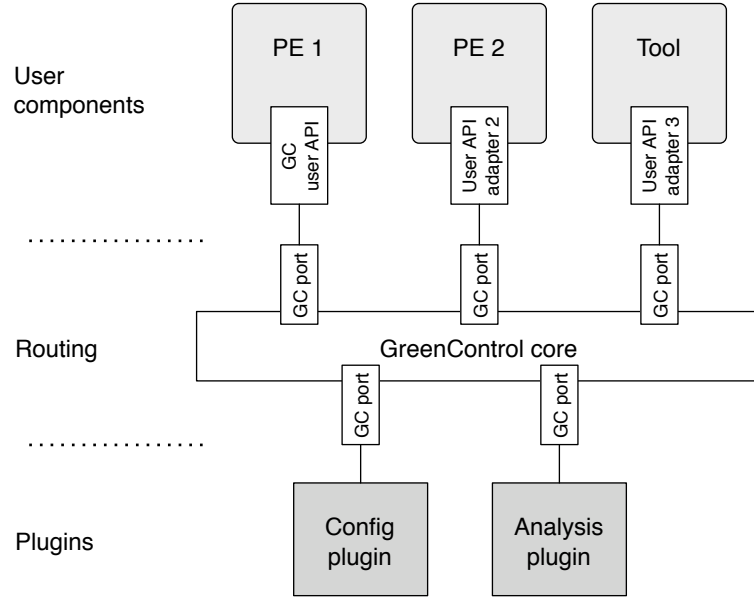


Figure 6.2: GREENCONTROL framework with service plugins

A survey of configuration libraries for SystemC revealed that none of them supports all these requirements. Table 6.1 gives an overview of the review results. The last column shows the capabilities of GREENCONTROL.

6.2.2 Transaction-based approach

The GREENCONTROL framework uses a transaction-based approach. The GREENCONTROL core is a router that connects PEs and tools with service providers (plugins). The connections are established via ports. Communication takes place using a generic transaction container. This concept is similar to GREENBUS PV transactions.

Figure 6.2 shows the approach. The GREENCONTROL core routes transactions between the PEs, an analysis tool, and plugins. A transaction container is used that contains generic attributes to transfer commands between the user modules and the service providers. Table 6.2 gives an example.

Both the router component and the generic ports of GREENBUS have been reused for the implementation. To ease usability, the required user APIs are automatically instantiated and connected to the router. The framework therefore traverses the simulation context before start of simulation and searches for unbound user APIs.

6.2.3 Configurable parameters

Listing 6.1 exemplifies the instrumentation of a PE with configurable parameters, using the ‘Green-Config API’, which has been developed as a simple interface to the framework.

Inside the PE, there is no difference in using configurable parameters or standard data types. The parameter data type is specified as a template parameter. From other PEs, the parameters can be accessed with the GreenConfig API.

6.2.4 Parameter database

All configurable parameters of a model are stored in a database. The database is managed by the ‘ConfigPlugin’. It only stores pointers to the configurable parameters, not their values, which makes

	CoWare SCML [27]	ARM CASI [9]	Texas Instruments SystemPython [11]	Intel DRF [104]	GreenControl
Data types	int, unsigned int, double, bool, std::string	std::string	strings and number formats	"Instrumented datatype" user implements parameter interface	PODs, SystemC types user types (implement interface)
User defined types	-	✓ (user implemented)	(unknown)	✓ (instrumented type)	✓ (template specialized)
Permanent storage	XML file	-	-	text file	text file
Global parameter registry	✓	✓	-	✓	✓
Set default value	✓ (constructor)	user may implement	user may implement	✓	✓ (constructor)
Get value during runtime by other PE or tool	✓	✓	✓	✓	✓
Change value during runtime by owner	✓	✓ (user implemented)	✓ (user implemented)	✓ (user implemented)	✓
Change value during runtime by other PE or tool	-	-	✓	-	✓
Notify value changes to user	-	-	✓ (task call)	✓ (event as member of instr. datatype)	✓ (register callback)
Notify value changes to tool	-	-	task-finished event	-	✓ (tool API registers callback)
Use wildcards / regular expressions for parameter access	-	-	-	-	✓

Table 6.1: This spreadsheet shows the capabilities of the reviewed configuration libraries for SystemC and compares them with the GREENCONTROL framework

Table 6.2: Example of a GREENCONTROL transaction container

Quark	Value
Service	ConfigService
Target	top.jpeg.compression_rate
Command	addParam
DefaultValue	42

Listing 6.1: Using configurable parameters in a PE

```

class bram
2 : public sc_module
{
4 protected:
    gc_param<uint32> size;
6    gc_param<uint8> operation_mode;
    gc_param<bool> dual_port;
8    gc_param<std::vector<uint8> > memory;

10 public:
    bram(sc_module_name name)
12     : sc_module(name),
        port("port"), // a GreenBus port (PE communication interface)
14        size("size", 0x200), // param name: "size", default value: 512
        operation_mode("operation_mode", BRAM_WRITE_FIRST),
16        dual_port("dual_port", false),
        data("data") // no default value
18    { [...] }

20    bool write(const std::vector<uint8> &data,
                const uint32 addr, const uint32 length) {
22        sc_assert(length<=size);
        memcpy(&memory[addr], &data[0], length);
24    }

26    [...]
};

```

parameter access very fast since no communication overhead is produced. Tools and PEs use the database to get access to parameters.

6.2.5 User APIs

Different user APIs have been implemented. For example, the SCML user API provides the interface methods defined by CoWare's SystemC Modeling Library (SCML) [27]. Thus, PEs that have been developed for the SCML library can be used with the GREENCONTROL framework.

6.2.6 Tool support

GREENCONTROL has been tested with a command line tool and CoWare's Platform Architect [26]. The command line tool has been written for the GreenConfig API. It provides a user console during simulation into which the user can type simple commands such as **set**, **get**, and **list** to inspect and set the configurable parameters in the design-under-test.

The Platform Architect tool from CoWare enables the analysis of models that use the SCML library. The SCML user API for GREENCONTROL implements the interface methods for parameter access of this library and thus allows for using Platform Architect with GREENBUS based models. Here it is interesting that GREENCONTROL not only can make Platform Architect believe it deals with SCML parameters, but also extends the capabilities of SCML parameters: with the command line tool SCML

parameters can also be set during simulation runtime although this functionality is not supported by CoWare's tool.

6.2.7 Testbench creation using configuration files

In addition to the configurable parameter framework a configuration file parser and a router fabric have been realized. The parser takes a configuration file such as shown in listing 6.2 and sets the parameters in the model accordingly. Several syntax formats are supported.

Listing 6.2: Example of a GREENCONTROL configuration file

```
1 # interconnect
  BUS(bus, PLB, 10, SC_NS)
3 CONNECT(master.port, bus.tport)
  CONNECT(bus.iport, bram.port)
5
7 # configure bram
  bram.size      8 # 8 byte
  bram.dual_port false
9  bram.data      { 0 42 200 12 64 0x2A 10 0xFF }
  bram.base_addr 0x1000
11 bram.high_addr 0x1008
13
15 # configure master
  master.target_addr 0x1000
  master.write_size  8 # write 8 byte per transaction
  master.read_size   2 # read 2 byte per transaction
```

Listing 6.2 also illustrates the capabilities of the router fabric for GREENBUS. The `BUS` command (line 8) instantiates the following interconnect components: the generic router, the PLB protocol class, and the PLB scheduler. Thus, a new CAFM is created. In the following `CONNECT` commands (lines 9-10) the two PEs `master` and `slave` are connected to this CAFM. The router fabric supports all CAFMs included with the GREENBUS kit. User CAFMs can be easily added (see source code documentation).

6.3 DUST: a SystemC-aware design analysis toolkit

To examine the potential of GREENCONTROL for performance analysis, the DUST³ toolkit has been developed in a series of student research projects ([15, 36, 102, 141]). DUST splits up into two parts. The backend collects and pre-process performance data. The frontend receives, stores, and visualizes the data. Figure 6.3 outlines this approach.

6.3.1 DUST backend

The backend adds analysis services to the design-under-test. Three service classes are distinguished:

- *Structure extraction*: this service explores the model components and provides GREENBUS-aware design structure information;
- *Monitor services*: they observe model properties using configurable parameters and collect performance information;
- *Injection services*: they inject control information into the design-under-test and can be used to implement debug features.

Access to the backend is provided by a network socket, using XML streaming.

³See [71] for a conceptual overview and discussion of related work.

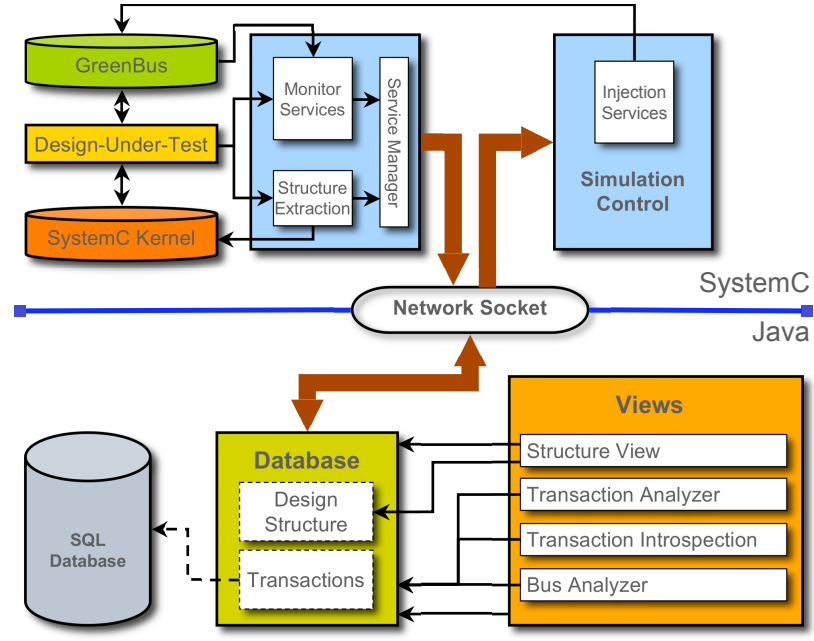


Figure 6.3: DUST analysis framework for SystemC

6.3.2 DUST frontend

The frontend has been implemented in Java. It establishes a network connection to the backend and communicates with the service manager. The service manager sends a list of the available services and allows for activation, configuration, and deactivation of services. Figures 6.4 and 6.5 show the frontend user interface. The following functionalities are provided:

- *Design structure analysis*

A graphical representation of the PEs and channels in the design-under-test is presented to the user. Model elements that do not add value to the visualization, such as inheritance trees (basic ports and parent classes) are omitted.

- *Performance data analysis*

The frontend can record and display the value changes of configurable parameters in the design-under-test. A message sequence chart plotter and atom tracing have been implemented (fig. 6.5).

- *Simulation control*

The SystemC simulation can be paused and the user can step to the next delta cycle or a given simulation time point. A future goal is to support conditional breakpoints.

6.3.3 Minimal-intrusive approach

An important goal of DUST was to enable performance analysis under the following constraints:

1. compliance with the SystemC 2.1 standard;
2. no changes are required to the SystemC simulation kernel source code;
3. only minimal code additions are required to enable analysis of a design-under-test.

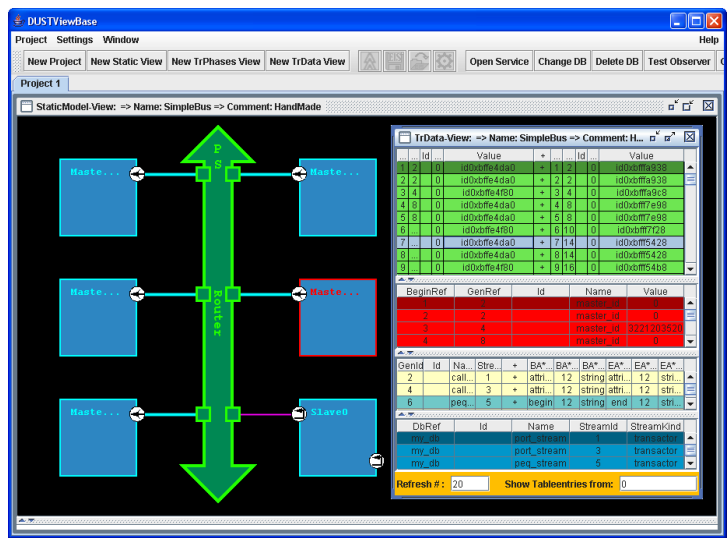


Figure 6.4: Design structure visualization and parameter monitoring

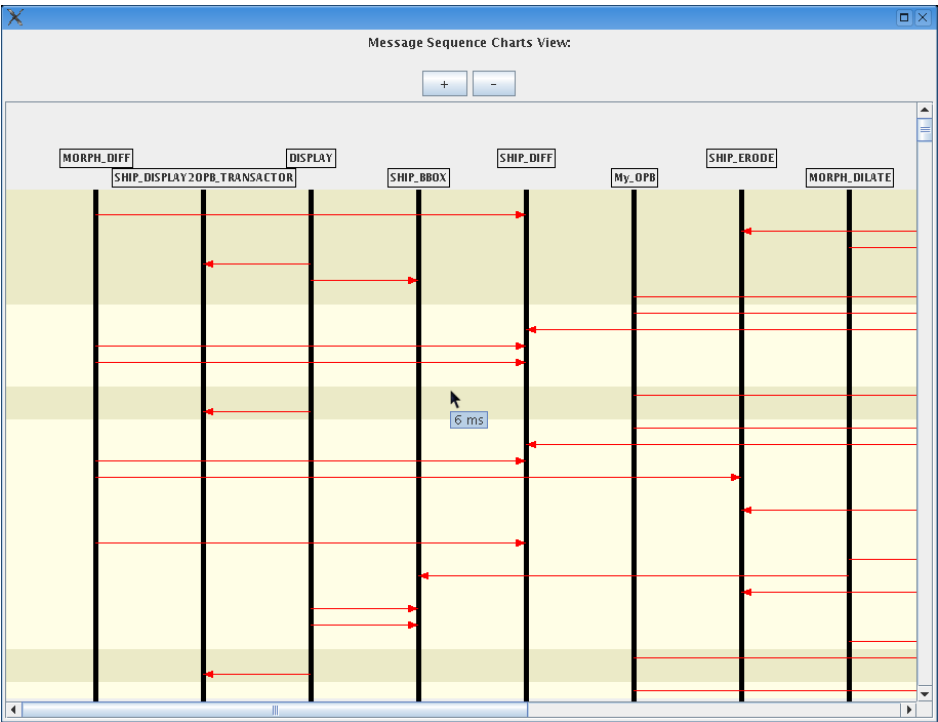


Figure 6.5: Message sequence chart view

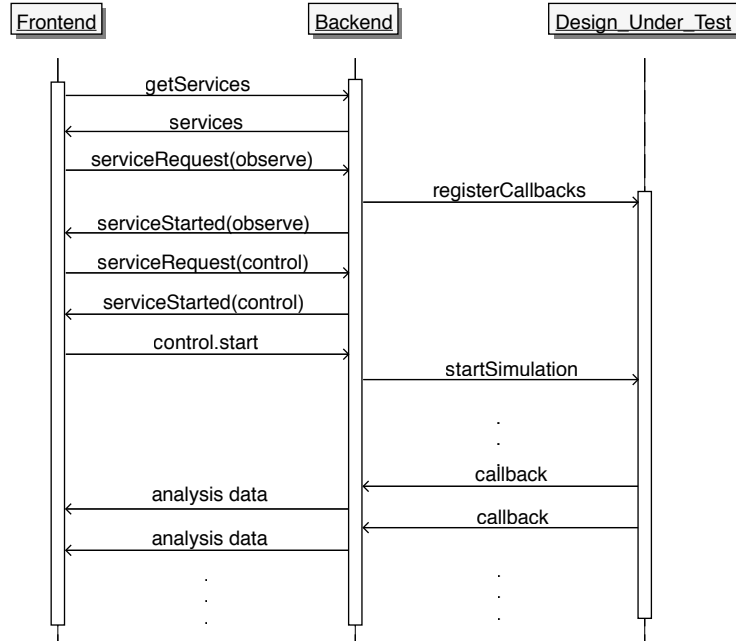


Figure 6.6: Communication between frontend and backend

DUST meets requirements 1 and 2 by using the `sc_trace` function to hook into the simulation kernel for simulation control. Requirement 3 can be fulfilled by using configurable parameters and dynamic linking. The GREENBUS implementation has been extended to support DUST analysis. To attach the DUST backend to the design-under-test, it can be passed to the C++ linker as a dynamic link library.

6.3.4 XML based data processing

The control flow and analysis data flow between frontend and backend is based on an XML format. Figure 6.6 shows an example data handshake. First, the frontend gets the list of available services in the design-under-test and invokes the observation service, thereby requesting to monitor some model parameters. Then the simulation is started using the control service. As a result, the frontend is informed about any value changes of the monitored parameters.

A formal definition of the XML format has been developed in XML schema [128]. With the aid of an XML parser generator (e.g., Apache XMLBeans [4]) Java and C++ classes can be generated to interpret XML data that is compliant to these schemas. Our experiments have shown that in connection with an SQL database this approach makes it easy to process and visualize large amounts of analysis data in Java. In [15] a tool has been developed that can generate the necessary SQL database tables from the XML service descriptions.

6.3.5 Compatibility

Experiments have shown that the DUST backend can be made compatible to commercial visual analysis tools that use SCV for data capturing, such as Mentor ModelSim [87] and CoWare Platform Architect. To make this work, we have implemented a new output plugin for GREENCONTROL that wraps the performance data from the design-under-test into an SCV stream. However, as the tested tools only support passive visualization, most capabilities of DUST lie idle when using such a tool. Figure 6.7 shows a screenshot of ModelSim displaying GREENBUS transactions.

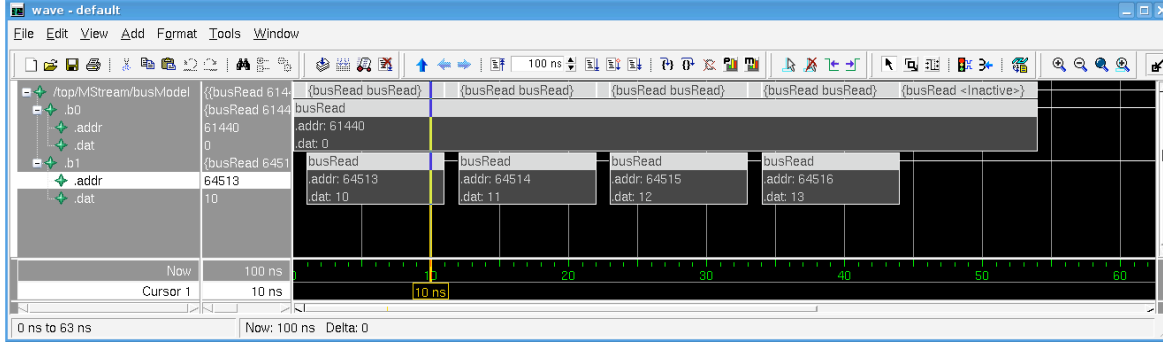


Figure 6.7: ModelSim displaying GREENBUS transactions using SCV

6.4 Summary and outlook

In this chapter, an analysis and debug framework for GREENBUS has been presented that enables introspection and reflection of different SystemC model properties using the concept of configurable parameters. GREENCONTROL allows for communication analysis and debugging without requiring changes to the design-under-test. The DUST toolkit demonstrates how GREENCONTROL can be used to enable visual analysis of GREENBUS-based models and SystemC-aware debugging. Furthermore, it has been shown that industry standard tools such as ModelSim can be connected, using the same layered approach (user APIs) as in GREENBUS for integration of heterogeneous IP. However, these user APIs must be implemented manually and their complexity depends on the tool interface. Further research is required whether a set of standard IO templates (e.g., SCV stream output, comma-separated list file output, gdb interface support) can help to ease this process.

In conclusion, GREENCONTROL contributes to the last two remaining requirements R2.12 and R2.13 from table 4.1 (page 34), see table 6.3.

Table 6.3: Fulfillment of user-view requirements from table 4.1 by GREENCONTROL

Req.	Description	Proposed technique	Implementation	Section
R2.12	Analysis & debug	Model instrumentation and observation using configurable parameters	GREENCONTROL framework based on analysis & debug transactions over a modified GREENBUS router	6.2
R2.13	Interfacing to external tools	Tool user APIs and service plug-ins	Layered, transaction-based approach with standardized low-level API	6.2.2, 6.2.5, 6.2.6

Conclusion

7 Summary and Future Work

In this thesis a standardization proposal for transaction level communication modeling with SystemC has been presented that eases the integration of heterogeneous hardware/software models and tools. A generic TLM fabric, GREENBUS, has been developed for proof-of-concept and it has been shown that this approach enables modeling and exploration of state-of-the-art embedded system and system-on-chip communication architectures at different, mixed levels of abstractions. The layered interface architecture of GREENBUS allows engineers to always use the communication APIs and tools that best support their specific needs and their specific style.

After an introduction (chapter 1), the following chapters 2 and 3 gave an overview of several approaches to transaction level communication modeling in SystemC, and identified from commercial bus fabrics and from discussions in the TLM community the requirements for GREENBUS, aiming at both model interoperability and clearness about the quality of simulation results. The discussion of the related work showed that there are several different and even incoherent interpretations of TLM and related abstraction levels. As a result, models developed with different frameworks and tools cannot be efficiently integrated, as their interfaces are not interoperable. My reviews revealed that in order to achieve interoperability across different interfaces and abstraction levels, the part of TLM communication that in fact needs to be standardized is surprisingly small. I suggest terms for these aspects (transaction, atom, quark, payload events, low level API) and propose a new TLM communication abstraction level, namely BA (bus accurate), which allows to create models with both cycle-count accurate (CC) and cycle-count accurate at transaction boundaries (CCATB) simulation accuracy. Chapter 4 developed and compared different implementation techniques for BA models and the outcome is the GREENBUS fabric for SystemC, with which PV, BA, and RTL models can efficiently inter-operate, independent of their respective interfaces.

Memory-mapped buses and also networks-on-chip can be directly mapped onto the proposed TAQ scheme (transaction, atom, quark), and in chapter 5 a generic router for GREENBUS has been presented which eases the implementation of BA functional models (CAFMs) for these communication architectures. Although being a generic solution, experiments with different architectures – including IBM CoreConnect OPB and PLB, PCI Express, and an Open Core Protocol based network-on-chip – showed that the achieved simulation performance can easily compete with the existing dedicated functional models for these architectures; in some cases it is even better.

To support the development of user APIs for GREENBUS, a generic protocol has been proposed. My implementation of well-known APIs for GREENBUS, such as ST Microelectronics TAC and OCP-TL1, shows this protocol to be quite versatile. In an industry cooperation, a PCI Express API has been developed. All these user APIs are inter-operable. Further user API implementations as well as a set of generic user API design patterns and state machine templates are presented in the appendix. These experiments show the BA abstraction level to be a sufficient approach for mixing both blocking and non-blocking, synchronous and asynchronous, untimed and timed communication in a single TLM model.

In addition to GREENBUS, concepts for advanced communication analysis and debugging have been presented in chapter 6:

- GREENCONTROL extends SystemC with ‘configurable parameters’ to enable performance analysis, SystemC/transaction-aware debugging, and tool-driven testbench creation.
- The DUST Java toolkit for visual model analysis has been developed by students and demonstrates the capabilities of GREENCONTROL.

The GREENBUS implementation and also the presented tools are still prototypic. However, ESL modeling groups at Intel Corp., Phoenix, and Texas Instruments, Nice, are already using them in their designs and have created own user APIs and CAFMs. The GreenSocs open-source initiative suggests the generic router in combination with GREENCONTROL as standard proposal for TLM-based architecture exploration and test, so the next step is to adapt GREENBUS to the changes and additions that have been made by the OSCI TLM Working Group to our proposal, such that it will adhere to the ‘official’ SystemC TLM 2.0 standard once it is released.

Several research areas are touched upon in this work. They can be divided into four main categories:

- communication modeling techniques and languages;
- communication architecture simulation and exploration;
- synthesis of such architectures;
- (physical) concepts for communication architectures.

While working on and experimenting with GREENBUS, I focused on the first two aspects. I could show that the concepts behind GREENBUS ease communication architecture exploration and design reuse, and the proposed abstraction level formalism helps to make TLM more systematic in general. But the presented concepts still require manual implementation of the user APIs, protocol classes, and schedulers. Hence, a key aspect of future research is to explore methodologies for automation of these tasks. One approach for the generation of application-specific user APIs is to generate them out of IP interface description. In [75], such an approach has been outlined for the special case of software user API generation. But this approach still requires a considerable amount of manual coding, so more advanced techniques based on formal interface descriptions should be examined.

GREENBUS has first and foremost been developed for high-level architecture exploration with abstract IP models. The intention is that these have been ‘extracted’ from existing RTL models, so when a suitable system configuration has been identified, they can be replaced with their RTL archetypes. Mixed-mode simulation with RTL IP is possible as well (using appropriate RTL user APIs), but this naturally slows down simulation performance significantly. The potentials and limitations of using GREENBUS in such a design flow have been investigated in a comprehensive case study: TLM design, architecture exploration, and FPGA prototyping of a real-time video processing system-on-chip for gesture recognition. Appendix C summarizes the results of this case study.

Another important aspect is further automation of communication architecture simulation and exploration. The generic protocol of GREENBUS presents the possibility of automatically generating protocol classes and schedulers for communication architecture functional simulation at BA abstraction from a description of the architecture features. For example, the SPIRIT [117] interface format could be extended into this direction. While visual analysis and debug tools are nice to have, and GREENCONTROL enables their seamless integration with GREENBUS, every designer at some time comes to the point where he needs to switch to the full power of manual test fixture programming. Here, script languages such as Perl and Lua can add great value. Hence the integration of scripting support into SystemC is a very interesting approach. In an industry-funded research project, GREENCONTROL is currently being extended into this direction.

The BA abstraction level approach discussed in this thesis mainly aims at systematic abstract simulation of RTL communication timing. Timing is definitely the most important issue in embedded system communication, but other characteristics are of interest as well, in particular power consumption. Hence the transaction-atom-quark approach needs to be examined whether it is also suitable for power estimation.

Finally, GREENBUS is ‘just’ a simulation environment, it does not enable automagic communication architecture synthesis out of abstract TLM models. With the TRAIN approach, which has been presented in [70] and also was used in the case study (appendix C), concepts were outlined how a library of RTL adapters can help in mapping a GREENBUS model onto a predefined platform. But

this approach is not generally new, requires the appropriate adapters to be in place, and is limited to the capabilities of the available buses.

Here, the upcoming network-on-chips with packet-based communication usher in a new era in communication architecture design. NoC architectures are far more flexible than traditional buses and I believe they will be the key enabler to bridge the gap from abstract models to RTL. The success of the Internet has proven hybrid network architectures to be the silver bullet, and with the flexible yet systematic communication modeling approach presented in this work it has been explored how an important ingredient of tomorrow's 'LEGO-style' design environments for such systems could look like.

Appendices

A Characteristics of SoC Communication Architectures

Contents

A.1 Overview	119
A.2 On-chip buses	119
A.3 Inter-chip buses	121
A.4 Bridges	122
A.5 Network-on-chip	122
A.6 Open Core Protocol	122

A.1 Overview

In embedded systems there are two major types of communication:

1. *Intra-chip communication*: This covers all transactions that take place inside a system-on-chip. The on-chip communication architecture (*SoC interconnect*) typically comprises point-to-point links for high-speed connections and several memory-mapped buses that connect processor cores with memories and peripherals. A novel technology are networks-on-chip (NoC) that aim for using Internet methodologies on a chip, such as packet based communication, dynamic routing, and quality-of-service.
2. *Inter-chip communication*: This covers all transactions that take place between chips, either on a board or among distributed networked embedded systems, e.g., in a car. On-board communication usually is enabled by buses with protocols similar to those used on-chip, whereas for inter-system communication a variety of communication media is used, such as Ethernet, CAN, MOST, USB, FireWire, Bluetooth, WLAN, etc.

In the following an overview on the existing communication architectures is given.

A.2 On-chip buses

Prominent on-chip bus architectures are AMBA, CoreConnect, and STBus.

- The *Advanced Microprocessor Bus Architecture* (AMBA) has been developed by ARM for connecting their soft-core processors to memories and other SoC components. AMBA 3 defines a set of three bus protocols: AXI (Advanced eXtensible Interface) [7], AHB (Advanced High-performance Bus) [8], APB (Advanced Peripheral Bus) [6]. While AXI and AHB provide the characteristics to support high data throughput with simultaneous read and write transactions on a pipelined interconnect, APB aims at lower bandwidth for simpler SoC components, allocating less hardware resources.
- The *CoreConnect* bus architecture has been developed by IBM to support SoC-integration of their PowerPC processors. CoreConnect comprises three buses, PLB (Processor Local Bus) [56], OPB (On-chip Peripheral Bus) [55], and DCR (Device Control Register bus) [54]. The PLB

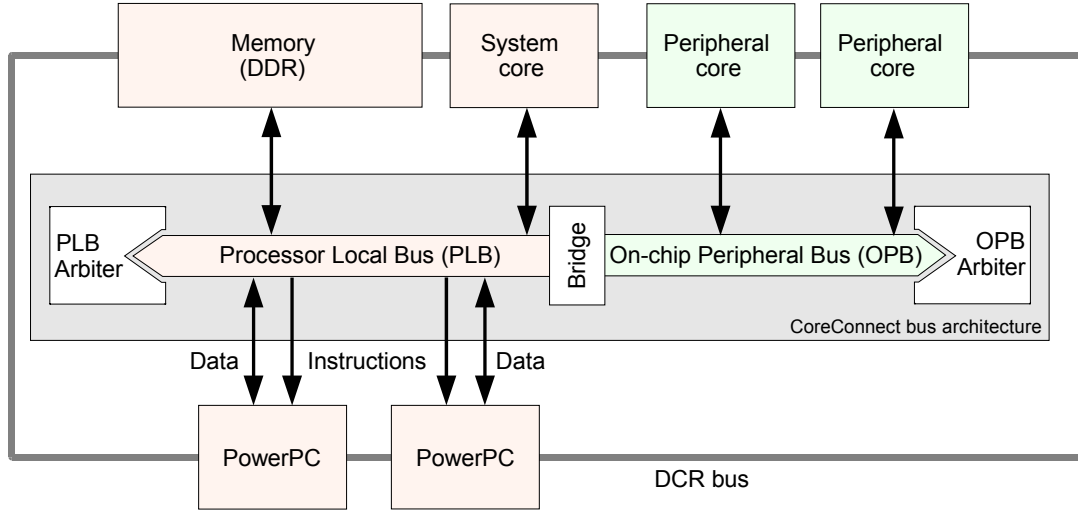


Figure A.1: CoreConnect architecture

enables simultaneous read and write transactions. It is intended to connect PowerPC processors to memories and SoC components with up to 256 bit wide data buses. For low-throughput communication between peripheral cores (e.g. Ethernet controller) the simpler OPB is used. The DCR daisy-chains all components on a chip to provide an alternate path to device control register. A block diagram of a typical CoreConnect architecture is shown in figure A.1. CoreConnect is used as a case example in this thesis to prove the applicability of the presented techniques. Its characteristics are comparable to those of AMBA and it is supported by Xilinx FPGAs, which I used for SoC prototyping in my research.

- *STBus* is an ST Microelectronics proprietary bus architecture. Contrary to AMBA and CoreConnect, STBus is a packed-based bus. Packets have a fixed size of at most 128 byte and are composed of cells that reflect the data bus width. Transactions are comprised of a request and a reply packet. Apart from that, STBus provides similar capabilities than AMBA and CoreConnect. Three protocols, Peripheral, Basic, and Advanced, are defined.

Characteristics common to the above on-chip buses are the following:

- *Memory-mapped*: Bus communication is memory-mapped. That is, each slave on the bus is given a range of memory addresses, masters use these addresses to specify the target of a transaction. An address bus is used to distribute the address information to the slaves.
- *Multi-master arbitration*: Most buses support multiple masters. Shared usage of the bus wires is controlled by a multiple access policy. Masters use control wires to signal bus requests. An arbiter continuously evaluates these wires and in case of multiple access it grants access using a scheduling algorithm. Simple scheduling algorithms are *round-robin* and *fixed-priority*, whereas more complex buses enable *dynamic priority* scheduling. When access has been granted, other masters are stalled until the transaction is finished. For example, PLB implements four priority levels of bus access, and masters can lock the bus for atomic operations, keeping out any other master until they have completed the locked transaction(s).
- *Combinatorial arbitration*: This is the property that master requests are evaluated combinatorial by the arbiter, so that the ‘winner’ is informed immediately. Data transfer can start in the same clock cycle. An example are the OPB and the PLB using single cycle arbitration.

- *Cycled arbitration*: This is the property that master requests are evaluated synchronous with the bus clock. The ‘winner’ is informed with the next clock cycle. Examples are the OPB and the PLB using two cycle arbitration.
- *Burst transfers*: Transactions are reads or writes that take place via a data bus of, e.g., 64 wires; that is, a data transfer of 64 bit. Burst transfers is the property that several words are transferred contiguously with one request. Bursts enable high transfer speeds. E.g., PLB pipelined back-to-back transfers enable transmission of one word per clock cycle.
- *Overlapping of read and write transfers*: PLB and AXI support two separate data buses, allowing for concurrent reads and writes, even for the same master.
- *Pipelined transfers*: With pipelined bus architectures, masters can issue requests while the current transaction is executing. This is enabled by using separate address and data buses, so that arbitration is overlapped with bus transfers, allowing data transfers to start immediately when the data bus gets available.
- *Out-of-order transactions*: If a bus device is unable to respond to a transfer initiated by another master until it has first executed a bus transaction of its own, it can make use of an ‘out-of-order’ transactions, allowing the device to have its request serviced ahead of the initial request. This feature is supported by both AXI and PLB¹.
- *Dynamic bus sizing*: This is the property that only parts of the data bus are used, e.g. for connecting a 32-bit slave to a 128-bit PLB.
- *Timeout*: To prevent deadlocks due to stalled slaves, the arbiter can cancel a transaction after a certain amount of idle cycles.

A.3 Inter-chip buses

Prominent buses for on-board communication are ISATM(Industry Standard Architecture) and its successors (EISATM, PC/104TM, etc.), PCITM(Peripheral Component Interconnect), and PCIe (PCI-Express). While ISA and PCI are shared buses, PCIe provides separate serial point-to-point links for each device. Characteristics of on-board bus protocols in many respects equal those discussed for on-chip buses, but usually provide less features to reduce the amount of control signals.

A wealth of buses is available for inter-system communication. USB (Universal Serial Bus), FireWireTM, FlexRayTM, TTPTM(Time Triggered Protocol), CAN (Controller Area Network), LONTM(Local Operating Network), and so on, are serial buses for data transfers between distributed devices via a minimal number of wires. While USB, and FireWire aim at high transfer speeds of several hundred MBit/s, FlexRay and TTP have been developed for hard real-time communication in cars and aeronautics. CAN is a very popular and cheap low-speed field bus for control message transfer, which is successfully used in automobiles since the 1980s. LON is an example for home automation buses (intelligent room climate control, etc.).

Common to all these communication architectures is that they support multiple masters (except USB), which is also true for the popular Ethernet. However, contrary to on-chip and on-board buses, arbitration takes place in a decentralized manner. That is, all devices that want to act as a master must agree upon a common media access policy to avoid collisions. Prominent access schemes are TDMA (Time Division Multiple Access) and CSMA (Carrier Sense Multiple Access).

¹Slave requested re-arbitration can be used to realize out-of-order transaction with the PLB.

A.4 Bridges

In a system with more than one bus, bridges can be used to enable masters on one bus to issue requests to slaves on another bus. In figure A.1, a PLB \Leftrightarrow OPB bridge is shown. Bridges are usually implemented as *transparent bridges*, such that they are not visible to the bus masters and slaves. They may contain an address map to perform address translation between different address ranges of the bridged buses. FIFO buffers may be necessary to enable bridging of burst transfers.

A.5 Network-on-chip

A special kind of bridges is used in network-on-chip architectures. The basic idea is to avoid the drawbacks of shared buses (weak performance under high bus load, complex arbitration) by using a network of point-to-point links that are connected by special bridges acting as routers. Transaction take place by setting up a path (logical link) through the NoC routers over which the data is transferred. Most NoC proposals use packet-based communication and static routing tables. Benefits of NoC methodology are:

- *Scalability*: a NoC can grow with the SoC;
- *Simplicity*: NoC interfaces can be kept simple because there is no complex arbitration;
- *Support for heterogeneous IP*: NoC routers can provide different interfaces to PEs, acting as protocol adapters.

However, NoC research is just at the beginning, and there are many problems looking for solutions. In particular, the switched architecture of NoCs can result in hardly predictable communication latencies, and the many routers and point-to-point links consume a lot of hardware resources. Hence, mixed NoC/bus architectures have been proposed. Recently, several NoC architectures have been proposed, see [12] and [64] for an overview.

A.6 Open Core Protocol

IP cores that have been developed for one of the above buses are not compatible to each other. Hence, IP reuse with another (processor) platform in a ‘plug-and-play’ manner is prohibited. A solution to this problem is to adhere to a bus-independent interface standard.

The most popular SoC interface standard is the Open Core Protocol (OCP) [94]. OCP has emerged from an existing proprietary standard (from Sonics, an interconnect IP company) and is now being promoted as ‘standard IP socket’ by an international industry partnership (OCP-IP [95]). Numerous IP vendors have joined this partnership and are developing IPs that ‘speak’ OCP. It provides a generic interface protocol with many configurable options, e.g. for fine-tuning burst transfer behavior or enabling out-of-order transfers. Thus, OCP can be customized by engineers to meet their requirements in terms of performance, range of functions, and interface complexity.

It has been shown that OCP cores can be connected to different buses using protocol adapters. For example, Prosilog provides OCP bridges for AMBA and CoreConnect [96], and ST Microelectronics developed OCP-to-STBus adapters [124]. In [13], OCP-based NoC modeling is proposed.

B User API Templates

Contents

B.1 Design patterns	123
B.2 State machines	127

In this appendix, design patterns and state machines are presented that can be used as a base for the development of custom user APIs for GREENBUS.

B.1 Design patterns

Interrupt			
Type:	Interrupt signals		
Use case:	P2P sideband signals		
Description:	Due to the finalization of the atom by the slave, the master gets informed when (and whether) the interrupt has been received. This allows to model propagation times (e.g., of an interrupt controller) and is useful for debugging purposes.		
Abstraction:	BA, CC (no differences)		
Atom sequence:	M.Request → S.AckRequest		
Quarks:	Name:	Set by:	Value:
	MBurstLength	Mst.	0
	MCmd	Mst.	Generic_MCMD_WR

Single beat write			
Type:	Write transaction of a single data word		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Transaction splits up into request and data phase. Thus, request arbitration time and data propagation time are estimated separately. The write data word is transferred with a single data atom. Its byte size depends on the data width of the master interface.		
Abstraction:	BA, CC (no differences)		
Atom sequence:	Mst.Request → Sl.AckRequest ⇒ Mst.SendData → Sl.AckData		
Quarks:	Name:	Set by:	Value:
	MAddr	Mst.	target address
	MBurstLength	Mst.	byte size of the data word
	MBurstPrecise	Mst.	true (default)
	MBurstSingleReq	Mst.	true (default)
	MSBytesValid	Mst.	same as MBurstLength
	MCmd	Mst.	Generic_MCMD_WR
	MData	Mst.	write data

Single beat read			
Type:	Read transaction of a single data word		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Transaction splits up into request and response phase. Thus, request arbitration time and data propagation time are estimated separately. The read data word is transferred with a single response atom. Its byte size depends on the data width of the slave interface.		
Abstraction:	BA, CC (no differences)		
Atom sequence:	Mst.Request \rightarrow Sl.AckRequest \Rightarrow Sl.Response \rightarrow Mst.AckResponse		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSingleReq MSBytesValid MCmd SData	Set by: Mst. Mst. Mst. Mst. Mst. Mst. Sl.	Value: target address byte size of the data word true (default) true (default) same as MBurstLength Generic_MCMD_RD read data

Single request multiple data (SRMD) fixed length burst write			
Type:	Fixed-length write transaction of multiple data words		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Transaction splits up into request and data phase. The data phase is comprised of a number of data atoms. Thus, request arbitration time and data propagation time are estimated separately, and the PEs can furthermore delay separate data chunks individually. The number of data words sent with each data atom is controlled by the master user API. The slave user API may accept less bytes than sent with a data atom, this is signaled back to the master user API with MSBytesValid. In such case, the non-accepted data words are re-issued with the next data atom.		
Abstraction:	At BA, typically all data will be transferred with a single data atom, but other segmentations are allowed as well. CC requires the master user API to send exactly one data word per atom (word width depends on master interface).		
Atom sequence:	Mst.Request \Rightarrow Sl.AckRequest \rightarrow Mst.SendData \Rightarrow Sl.AckData \rightarrow Mst.SendData \Rightarrow ... \Rightarrow Sl.AckData		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSingleReq MSBytesValid MSBytesValid MCmd MData	Set by: Mst. Mst. Mst. Mst. Mst. Sl. Mst. Mst.	Value: target address byte size of the fixed-length write data true (default) true (default) number of bytes valid with this atom number of bytes accepted with this atom Generic_MCMD_WR write data

Single request multiple data (SRMD) fixed length burst read			
Type:	Fixed-length read transaction of multiple data words		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Similar to SRMD fixed-length burst write, with the difference that the number of data words sent with each data atom is controlled by the slave user API. The master user API may accept less bytes than sent with a data atom, this is signaled by the value of MSBytesValid. In that case, the slave user API must re-issue the non-accepted data words with the next data atom.		
Abstraction:	At BA, typically all data will be transferred with a single data atom, but other segmentations are allowed as well. CC requires the slave user API to send exactly one data word per atom (word width depends on slave interface).		
Atom sequence:	Mst.Request \Rightarrow Sl.AckRequest \rightarrow Sl.Response \Rightarrow Mst.AckResponse \rightarrow Sl.Response \Rightarrow ... \Rightarrow Mst.AckResponse		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSingleReq MSBytesValid MSBytesValid MCmd SData	Set by: Mst. Mst. Mst. Mst. Sl. Mst. Mst. Sl.	Value: target address byte size of the fixed-length read data true (default) true (default) number of bytes valid with this atom number of bytes accepted with this atom Generic_MCMD_RD read data

Multiple request multiple data (MRMD) arbitrary length burst write			
Type:	Arbitrary-length write transaction of multiple data words, using dedicated requests per data word		
Use case:	P2P, memory-mapped buses, NoC		
Description:	MRMD transactions are modeled with a sequence of single-beat transactions. Usage of the MRMD protocol is indicated by setting the <code>MBurstSingleReq</code> quark to <code>false</code> . Note that <code>MBurstLength</code> and <code>MSBytesValid</code> are set to the number of bytes transferred with the <i>current</i> transaction. Thus, they do not reflect the overall burst length of the MRMD transaction. The last transaction of the MRMD burst write is marked by <code>MReqLast</code> . <code>MBurstSeq</code> can be (optionally) used to detect out-of-order transfers in NoC interconnects.		
Abstraction:	BA, CC (no differences).		
Atom sequence:	Mst.Request \rightarrow Mst.SendData \Rightarrow Sl.AckData		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSeq MBurstSingleReq MSBytesValid MCmd MData MReqLast	Set by: Mst. Mst. Mst. Mst. Mst. Mst. Mst. Mst. Mst.	Value: target address byte size of the write data of this transaction <code>true</code> (default) sequence number of this transaction in the MRMD burst <code>false</code> number of bytes valid with this <i>transaction</i> <code>Generic_MCMD_WR</code> write data for <i>this</i> transaction <code>true</code> for last transaction in the MRMD burst, otherwise <code>false</code>

Multiple request multiple data (MRMD) arbitrary length burst read			
Type:	Arbitrary-length read transaction of multiple data words, using dedicated requests per data word		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Similar to MRMD arbitrary length burst write transactions. The last transaction of the MRMD burst read is marked by <code>MReqLast</code> .		
Abstraction:	BA, CC (no differences).		
Atom sequence:	Mst.Request \rightarrow Sl.Response \Rightarrow Mst.AckResponse		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSeq MBurstSingleReq MSBytesValid MCmd SData MReqLast	Set by: Mst. Mst. Mst. Mst. Mst. Mst. Mst. Sl. Sl.	Value: target address byte size of the read data of this transaction <code>true</code> (default) sequence number of this transaction in the MRMD burst <code>false</code> number of bytes valid with this <i>transaction</i> <code>Generic_MCMD_RD</code> read data for <i>this</i> transaction <code>true</code> for last transaction in the MRMD burst, otherwise <code>false</code>

Single request multiple data (SRMD) arbitrary length burst write			
Type:	Arbitrary-length write transaction of multiple data words, using a single request		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Similar to SRMD fixed length burst, but with the difference that the master decides <i>during</i> the transaction how much data is to be sent. The slave user API must check <code>MBurstLength</code> for each data atom. If its value equals to <code>MSBytesValid</code> , the transaction is finished.		
Abstraction:	At BA, the master controls how much data to send per atom. CC requires the master user API to send exactly one data word per atom (word width equals to data port width).		
Atom sequence:	Mst.Request \Rightarrow Sl.AckRequest \rightarrow Mst.SendData \Rightarrow Sl.AckData \rightarrow Mst.SendData \Rightarrow ... \Rightarrow Sl.AckData		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSingleReq MSBytesValid MSBytesValid MCmd MData	Set by: Mst. Mst. Mst. Mst. Mst. Mst. Mst.	Value: target address Same as <code>MSBytesValid</code> for last data atom, otherwise 0 <code>false</code> <code>true</code> (default) number of bytes valid with this atom number of bytes accepted with this atom <code>Generic_MCMD_WR</code> write data (filled up during the transaction)

Single request multiple data (SRMD) arbitrary length burst read			
Type:	Arbitrary-length read transaction of multiple data words, using a single request		
Use case:	P2P, memory-mapped buses, NoC		
Description:	Similar to SRMD arbitrary length burst read. The master controls <i>during</i> the transaction when the slave is to stop sending read data. The slave user API must check MBurstLength for each response atom. If its value equals to MSBytesValid (including the read data sent with the current response atom), the transaction is finished.		
Abstraction:	At BA, the slave controls how much read data to send per atom. CC requires the slave user API to send exactly one data word per atom (word width equals to port width).		
Atom sequence:	Mst.Request \Rightarrow Sl.AckRequest \rightarrow Sl.Response \Rightarrow Mst.AckResponse \rightarrow Sl.Response \Rightarrow ... \Rightarrow Mst.AckResponse		
Quarks:	Name: MAddr MBurstLength MBurstPrecise MBurstSingleReq MSBytesValid MSBytesValid MCmd SData	Set by: Mst. Mst. Mst. Mst. Sl. Mst. Mst. Mst.	Value: target address set to the total burst length for the last response atom acknowledge, otherwise 0 false true (default) number of bytes valid with this atom number of bytes accepted with this atom Generic_MCMD_RD read data (filled up during the transaction)

B.1.1 MSBytesValid

The **MSBytesValid** quark plays an important role in generic protocol communication. When a data or response atom is issued, it indicates how many bytes of **MData** are valid with the current atom. When a data or response atom is finalized, it indicates how many valid bytes have been accepted by the target. Hence, during the data phase the value of **MSBytesValid** is read and potentially changed by both master and slave user API for each atom.

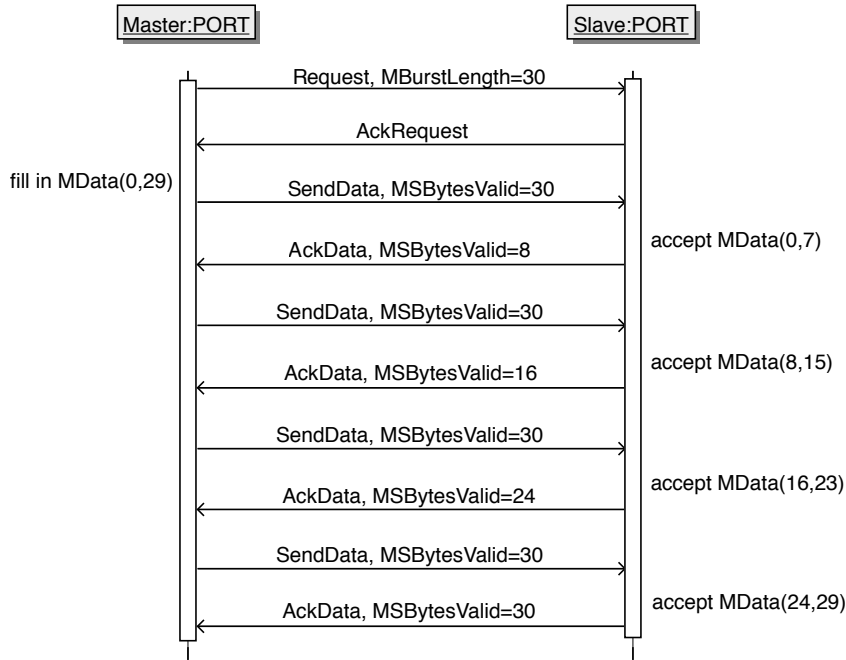
In write transactions, the master port informs the slave port with **MSBytesValid** how many bytes it wants to transmit with the current data atom. In BA mode, this typically will be the complete burst length. Lower values are permitted as well, which in arbitrary-length bursts makes sense; splitting the transaction into data chunks of application-defined length. In CC mode, the value of **MSBytesValid** set by the master user API *must* be less or equal to **MDataWidth**.

When a data atom is received by the slave user API, it checks **MSBytesValid** and considers how many bytes it is able to accept. In BA mode, that usually should be all bytes marked valid. The slave user API calculates an appropriate acceptance delay (see figure 4.8) and passes it to **AckData**. In CC mode, the number of accepted bytes depends on the width of the simulated physical data interface. In this case, the master must be informed, such that it can re-issue the ‘lost’ bytes and/or alert its PE. This is done by resetting **MSBytesValid** to the highest byte number that has been accepted by the slave. Hence, the master user API needs to check **MSBytesValid** upon reception of the **AckData** and, in case of data loss, must take actions needed (i.e., retransmit the lost data bytes).

To get a clear understanding of this approach, consider figure B.1. It shows the values of **MSBytesValid** for a 30 byte SRMD burst write. In this example, the master user API runs in BA mode, whereas the slave user API implements CC communication for a data interface width of 128 bit.

Read transfers are performed in a similar manner, but with the difference that the slave user API sets **MSBytesValid** to the number of bytes valid with response atoms and the master user API may change **MSBytesValid** to a lower value if it cannot accept all data bytes.

This mechanism enables communication among PEs and routers with different interface widths and at different abstractions. Note that its implementation is very simple, as there is no difference between BA \leftrightarrow BA and BA \leftrightarrow CC communication. Listing B.1 shows an extended version of the simple TAC master user API example from listing 4.8 on page 70. While the version from page 70 only supports the single-beat transaction design pattern, this version also supports SRMD burst writes to any BA or CC slave, thereby using **MSBytesValid** to prevent data loss due to fragmentation (lines 20 to 25).

Figure B.1: Usage of `MSBytesValid` in generic protocol transactions

B.1.2 Slave terminated transaction types

In addition to the transaction types shown above whose burst length in all cases is under control of the master, some communication architectures allow the slave to actively terminate a transaction. For example, on the Processor Local Bus (PLB) a slave can set the `S1_wrBTerm` signal to terminate a write burst (see [56]). This functionality is not covered by the standard phases of the generic protocol.

The motivation for this decision is to keep the generic protocol as simple as possible. The complexity of user APIs and their development effort increases with each new feature added to the generic protocol. Hence, I followed a minimalist approach with the goal to only support the basic transaction types of the four major bus families natively – namely AMBA, CoreConnect, STBus, and Wishbone – while user extensions (see 4.8) enable the implementation of advanced protocol features. For instance slave active transaction termination can be modeled by adding a new user atom, `AckDataAndTerm` (see 5.4.1).

B.1.3 Error handling

The atom sequences specified with the above design patterns do not explicitly consider error handling. However, all generic protocol calls with the prefix ‘Ack’ can be replaced with an error notification by using the prefix ‘Error’. Thus, all components involved in a transaction are able to abort it with an error at any time point of communication. The `SError` quark can be used to explain the error reason.

An error notification always finalizes the transaction. That is, any outstanding atoms of the transaction are cancelled, and all components involved into the transaction reset themselves into an idle state.

B.2 State machines

Figures B.2 and B.3 show state machines for the realization of read and write transfers in master ports, applying the above design patterns. The arrows in the graphs denote state transitions and are

Listing B.1: Extending the TAC master port write method for using MSBytesValid

```

1  tac_status write(const uint32& address,
2                  const std::vector<byte>& data,
3                  tac_error_reason& error_reason)
4  {
5      transactionHandle th = mst_port.create_transaction();
6
7      // map TAC attributes onto quarks
8      th->setMCmd(Generic_MCMD_WR);
9      th->setMAddr(address);
10     th->setMData(data);
11     th->setMSBytesValid(data.size());
12
13     GenericPhase ph;
14
15     // send request atom
16     ph = PORT::Request.block(th);
17     if (ph.isRequestAccepted()) {
18         // send data atom and handle potential data fragmentation
19         ph = PORT::SendData.block(th, ph);
20         while (ph.isDataAccepted()
21               && th->getMSBytesValid() < data.size()) {
22             th->setMSBytesValid(data.size());
23             // send next data fragment
24             ph = PORT::SendData.block(th, ph);
25         }
26     }
27
28     // set TAC error state
29     tac_status status;
30     if (!ph.isDataAccepted()) {
31         status.set_error();
32         error_reason.set_reason(ph.toString());
33     }
34     else
35         status.set_ok();
36     return status;
37 }

```

labeled with the event (if any) causing this transition and an optional guard expression enclosed in square brackets. If an action is performed during a transition, it is added to the label following a ‘/’. The state machines support single-beat, SRMD, and MRMD transactions with fixed burst length at both BA and CC abstraction.

It can be seen that the realization of BA and CC communication does not fundamentally differ with Greenbus – CC transaction are equal to SRMD transactions with sending one data atom per data word, whereby the word bit width depends on the bit width of the PE data interface. Figures B.4 and B.5 show the state machines for the realization of appropriate slave user APIs.

B.2.1 Implementation

Using these state machines as a base, two methodologies are conceivable for the implementation of user APIs.

First, the state machines can be implemented *implicitly* using a sequence of blocking generic protocol method calls. This approach is appropriate for simple high-level user APIs such as TAC and results in code that is easy to understand, but will fail if advanced functionality is required such as pipelined transactions.

More complex user APIs therefore require an *explicit* implementation of the state machines, using concurrent SC_METHOD processes and additional internal events and variables to propagate and store the current state. This may result in larger code size, but the code gets distributed among several

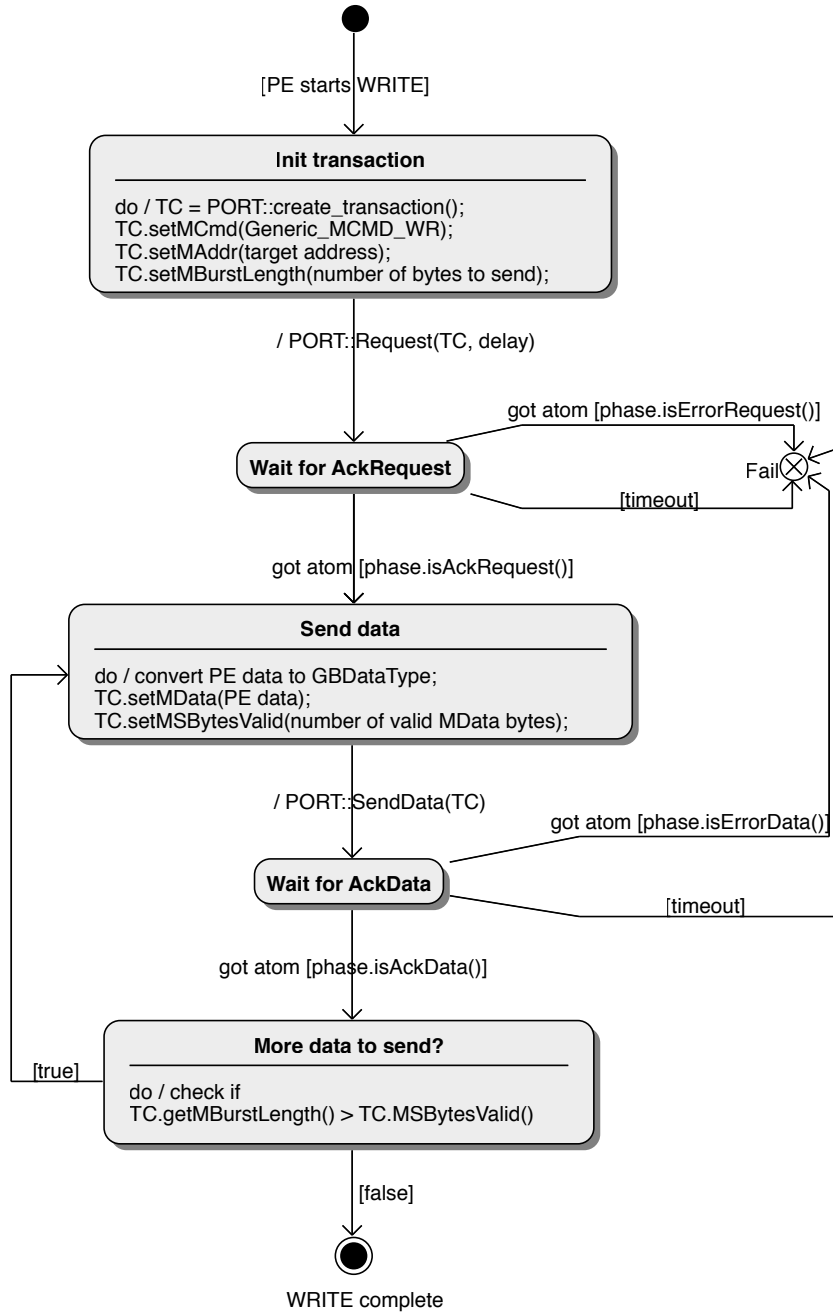


Figure B.2: State diagram for master user API; write transaction with fixed burst length (both single-beat and SRMD)

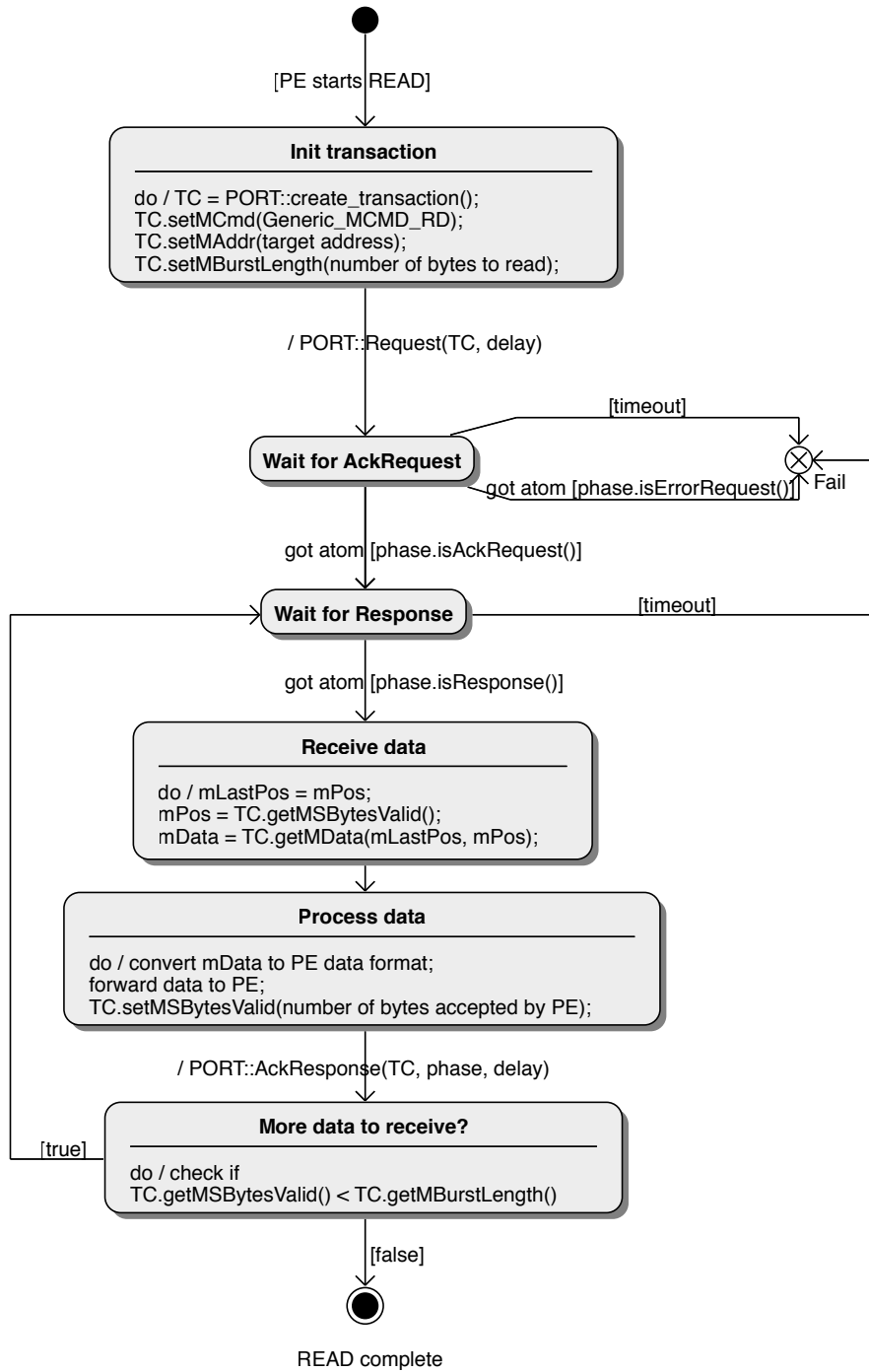


Figure B.3: State diagram for master user API; read transaction with fixed burst length (both single-beat and SRMD)

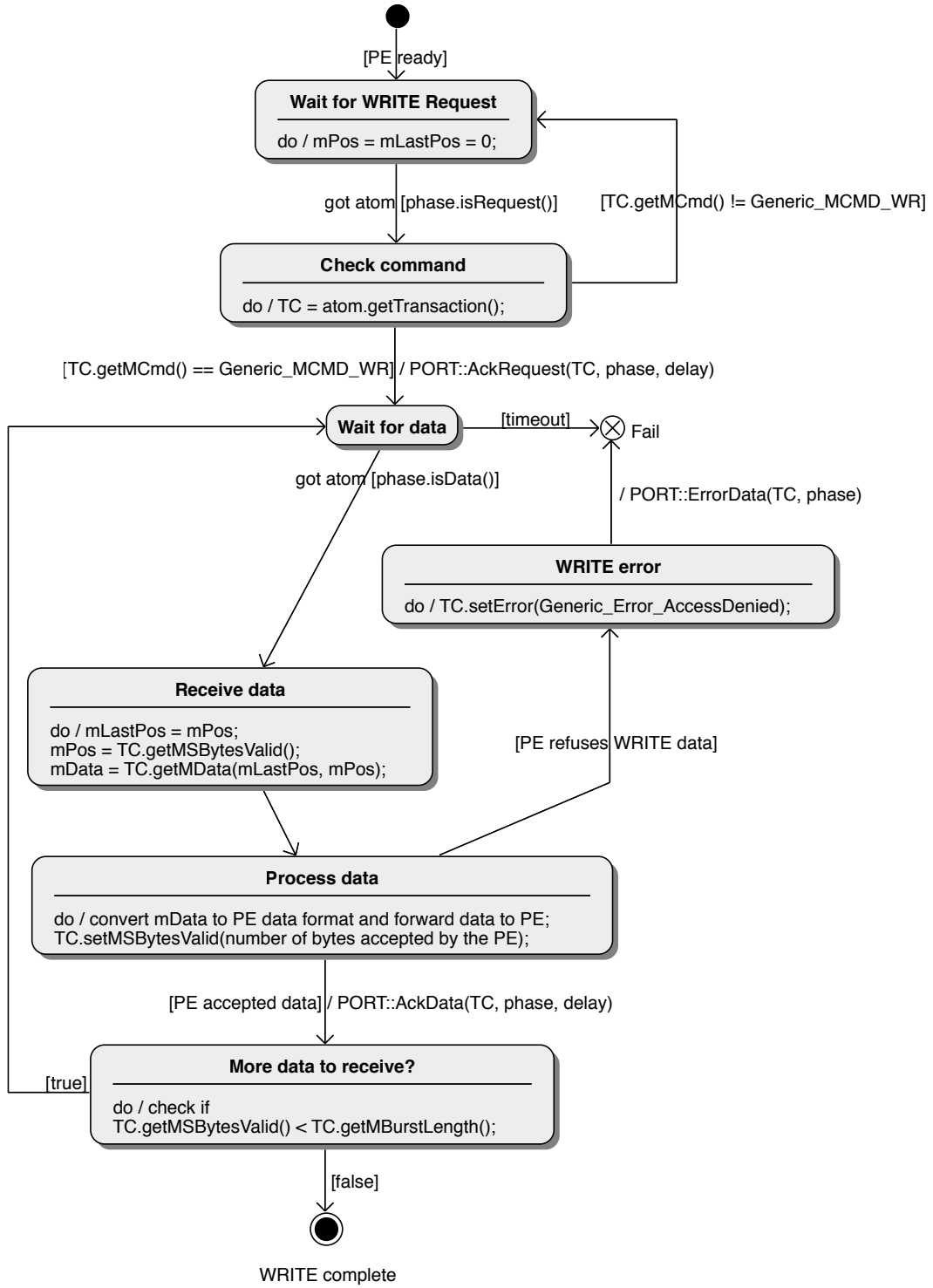


Figure B.4: State diagram for slave user API; write transaction with fixed burst length (both single-beat and SRMD)

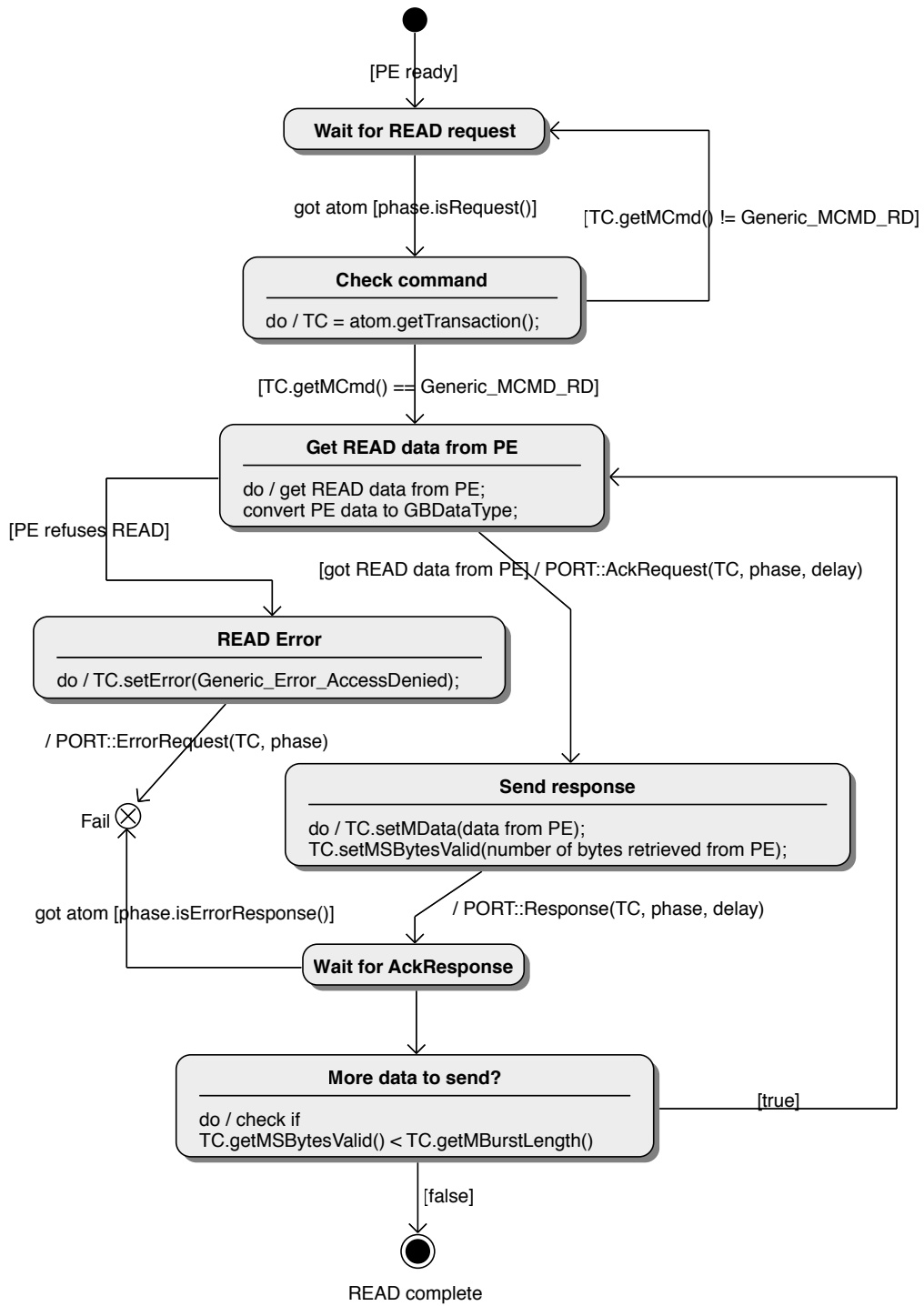


Figure B.5: State diagram for slave user API; read transaction with fixed burst length (both single-beat and SRMD)

(potentially concurrently executed) methods. Each of them realizes only a small part of the functionality, which contributes to code quality and maintainability. In both approaches, the `notify` method listens for incoming payload events and acts as a dispatcher, triggering port-internal events to initiate processing of the received atom.

The shown state machines cover only the basic functionality of generic protocol communication. Usually, the designer will add further (sub-)states for data conversion, error recovery, packeting and de-packeting, debugging, analysis, and so forth. Additional quarks may be used such as `MThreadID` and `MTagID` to enable overlapping transactions and modeling of multi-threaded software.

The visibility of the port internal states to the PE depends on the abstraction of the PE interface.

High-level interfaces such as TAC and SHIP will only indicate the completion (or abortion) of a transaction to the PE, such that the whole state machine is implemented in the port. Here, an implicit state machine implementation is sufficient, since there is no interaction between user API states and PE processes. This has been exemplified with the TAC API in the last section.

For lower level PE interfaces, the state machine implementation typically will be distributed among the PE processes and the port. For example, OCP-tl2 requires the PE to actively wait for the acknowledgment of its transaction requests, and OCP-tl1 furthermore requires the PE to actively push sending of each individual data word.

B.2.2 PV transactions

Due to their simplicity, PV transactions have not been addressed as yet. PV transactions should be exclusively modeled with the blocking `transact` interface. All information related to the transaction is transferred with the transaction container – there is no transaction state information (i.e., atoms) required.

To exercise a write transaction, the master port fills in all write data *a priori* and then calls `transact`. For a read, the read data is copied into the transaction container by the slave user API `transact` implementation. Instead of actually converting and deep-copying user data into the `GBDataType` data format, in PV mode it in many cases is sufficient to simply pass a data pointer through the interconnect. This is safely possible because PV method calls are always blocking.

B.2.3 Abstraction switching

Master user APIs can provide a configurable parameter to support toggling the preferred abstraction mode. Thus, the PE (or a tool) can control whether to perform PV, BA, or CC transactions.

C EmViD Video Processor Case Study

Contents		
C.1	Overview	135
C.2	High-level modeling and refinement with SHIP	136
C.3	TRAIN and HPC	142
C.4	Architecture exploration and synthesis of a real-time gesture recognition system	144
C.5	Conclusion and outlook	154

C.1 Overview

In this appendix it is shown how GREENBUS and the associated tools can be adopted for systematic embedded system design. Starting with the golden model, the design flow is split up into concurrent processes for software development, hardware development, and IP integration (fig. C.1).

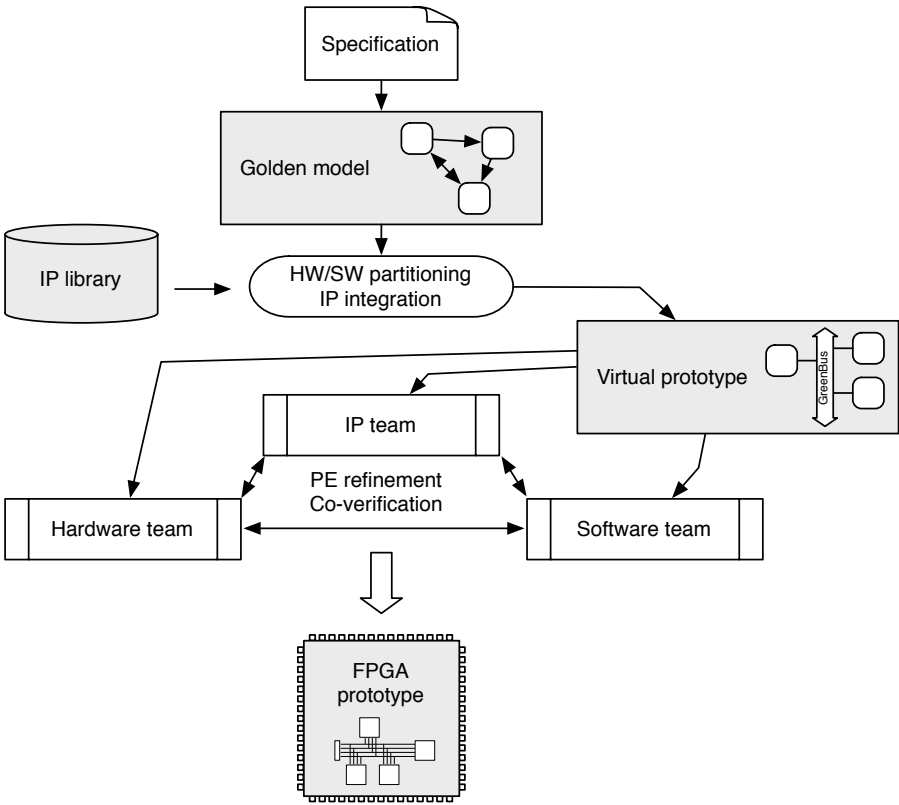


Figure C.1: Systematic embedded system design flow

In [70, 72, 75], the following three approaches around GREENBUS were presented with the goal to relieve the TLM engineer of the task of implementing platform-specific communication protocols:

- SHIP (SystemC High-level Interface Protocol), a synchronous message-passing protocol, helps to create deterministic golden models whose communication interfaces can be semi-automatically refinement [72];
- TRAIN (Transaction Interchange), a library of generic communication adapters, eases mapping of SHIP channels onto a target platform, including HW-SW interface generation [70];
- HPC (Hardware Procedure Calls), a GREENBUS ‘middleware’ for architecture-independent software development [75].

We applied these approaches to a complex case example, the Embedded Video Detection (EmViD) system-on-chip platform, using GREENBUS as central communication fabric and GREENCONTROL plus DUST for analysis. EmViD consists of a set of hardware (SystemC, VERILOG) and software (SystemC) cores that can be integrated to build application-specific video processors. Out of high-level PEs which I wrote in SystemC, several hardware cores have been refined to RTL in student projects ([45, 50, 52, 89, 110, 121, 122, 140]). These include IP for video input (from a camera) and output (to a VGA display), picture noise reduction, color balancing and conversion, edge accentuation, region and motion detection, as well as a JPEG encoder.

By assembling the video IP cores (‘VICs’) in different manners, various video applications can be realized. In the following, different EmViD examples are used to examine the potentials and limitations of using GREENBUS in the TLM design methodology, and finally an FPGA prototype of a complex gesture recognition system is constructed, using all tools from this PhD thesis concertedly.

C.2 High-level modeling and refinement with SHIP

SHIP (SystemC High level Interface Protocol) is a lightweight communication protocol for directed synchronous message-passing between two PEs. SHIP was designed to allow for top-down communication modeling on a level of abstraction that is completely independent of the HW/SW partitioning.

Four blocking interface methods are offered: `send`, `recv`, `request`, and `reply`. They transfer any C++ object that implements the `gb_serializable_if` interface `serialize` and `deserialize` functions. Objects that fulfill this requirement are called *SHIP objects*.

While PEs that exclusively use the `send` and `request` methods implicitly represent a communication master, `recv` and `reply` are slave methods. When consequently applied, this allows for automatic master/slave detection.

SHIP is untimed. Synchronization is achieved with the following interface semantics: a master `send` call returns if and only if the slave `recv` method received the SHIP object and returned; a master `request` call returns if and only if the slave called `reply` and the passed SHIP object has been received. This communication style is depicted in figure C.2.

The synchronous blocking communication enables to describe a system functionality by a number of function blocks (PEs) connected with SHIP channels. The resulting model of computation has similarities to sequential communicating processes [51]. In particular, there is no explicit timing information required in the model. Deterministic (partial) process execution orders are assured by the synchronization points which are determined through the blocking SHIP method calls.

Example 3. Figure C.3 shows the usage of SHIP in the EmViD platform. For each video processing component a function block with a SHIP slave and master port has been developed. The internal process waits for an incoming video frame on the slave port by calling the blocking `recv`, then processes the frame (apply a morphing algorithm) and sends the result to its output by calling `send` on the master port.

Three SHIP objects are used (fig. C.4). `YUVFrame` transports color images using the YUV 4:2:2 color space (2 byte per pixel, see e.g. [106]), `GRAYFrame` transports gray images (1 byte per pixel), and `BINFrame` transports black-and-white images (1 bit per pixel). All three classes inherit from the

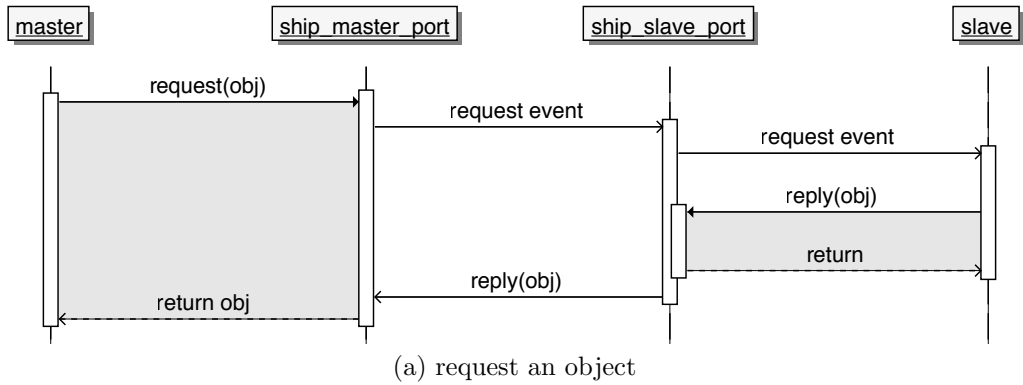
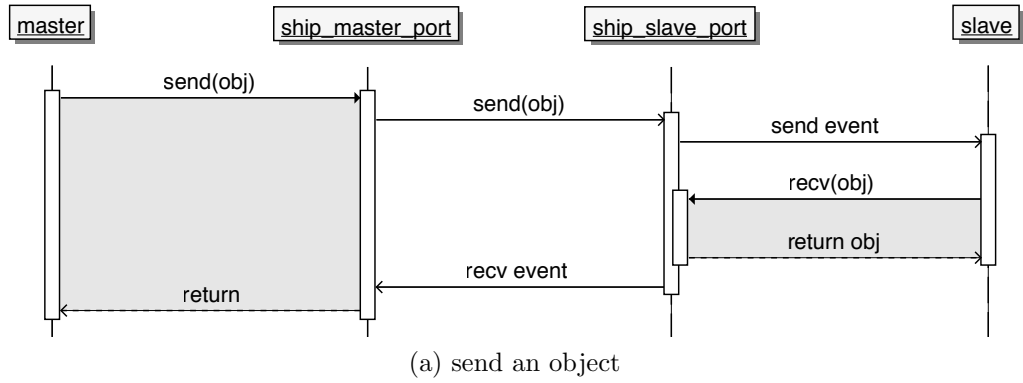


Figure C.2: SHIP communication

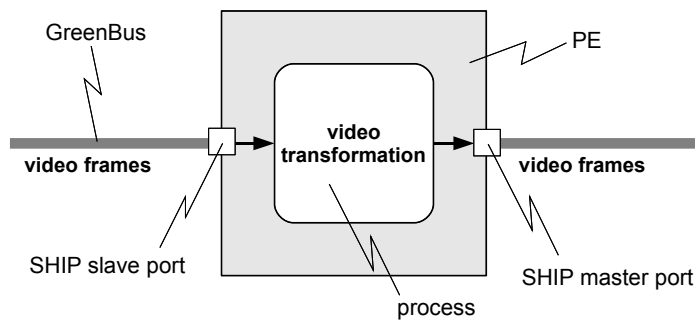


Figure C.3: EmViD abstract PEs use SHIP communication

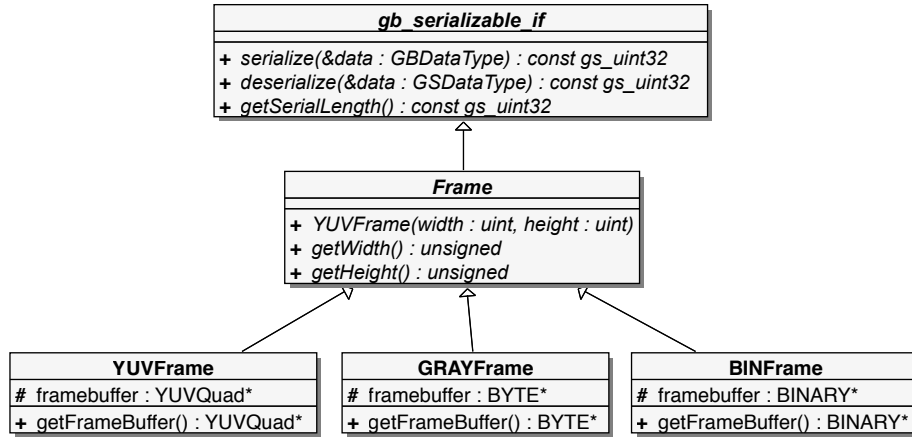


Figure C.4: SHIP objects used in EmViD for video frame processing and transport

same base to allow for implicit image conversion and implement the `gb_serializable_if` interface for usage with SHIP. Access methods such as `getFrameBuffer` or `setPixel` are provided to enable an object-oriented modeling style (note that the full list of access methods is omitted in the class diagram).

C.2.1 GreenBus implementation of SHIP

I have implemented SHIP for GREENBUS in the form of two user APIs: the master commands `send` and `request` are implemented in the `shipMasterAPI`, the `shipSlaveAPI` provides the `recv` and `reply` commands as well as a `waitEvent` method. The latter allows a slave to handle both `send` and `request` commands in a single process. It waits for an incoming master request and returns the request type. Then the slave must call either `recv` or `reply`.

In PV mode, the SHIP object is not serialized. Only a pointer to the object is passed with the `MData` quark of the transaction container, and the blocking `b_transact` is used (cp. 4.5). This achieves very high simulation speeds. For BA and CC transactions the object is serialized.

SHIP was designed as a point-to-point protocol and therefore does not consider address information. Nevertheless communication architecture exploration with GREENBUS is possible: the SHIP master port contains a configurable parameter with which a default target address can be set, and for the SHIP slave port an address range can be specified (see 4.6.3.2). Thus, SHIP ports can be connected to a GREENBUS router (see 5.1), and they can exchange data with any other user API that adheres to the generic protocol (see 4.8.2).

Example 4. From the *EmViD* executable specification, a golden model is derived by distributing the functionality over a number of PEs. Figure C.5 shows a video processing pipeline for gesture recognition. It consists of a number of video processing components that exchange data and synchronize via SHIP (each arrow in the picture is a GREENBUS master port to slave port binding). Note that no decision according the HW/SW partitioning has been made yet.

Given a library of video processing components, the pipeline behavior can be changed by inserting, removing, or reordering components. Table C.1 lists the components required for gesture recognition. They use the above described data types. Some support multiple video formats and can be configured to the appropriate data type. Only `RegionDetect` is an exception: it returns a list of region coordinates (as a corresponding `RegionList` SHIP object) when used in connection with a `Labeling` component.

In [110] a Java tool for the graphical composition of video processors has been developed. Fig. C.6 shows a screenshot. The tool generates SystemC code that can be directly fed into the compiler. The result can be used as a virtual prototype for functional verification and architecture exploration with GREENBUS and DUST.

Table C.1: SHIP-based video processing components for gesture recognition

Component	Input	Output	Description
Color matching	YUVFrame GREYFrame	GREYFrame BINFrame	Remove pixels that do not match a given color range
Erosion	YUVFrame GREYFrame BINFrame	YUVFrame GREYFrame BINFrame	Remove noise
Dilation	YUVFrame GREYFrame BINFrame	YUVFrame GREYFrame BINFrame	Merge adjacent pixel groups
Labeling	YUVFrame GREYFrame BINFrame	GREYFrame	Label cohesive pixel groups
Region detection	GREYFrame	RegionList	Calculate transitive closure of labeled regions

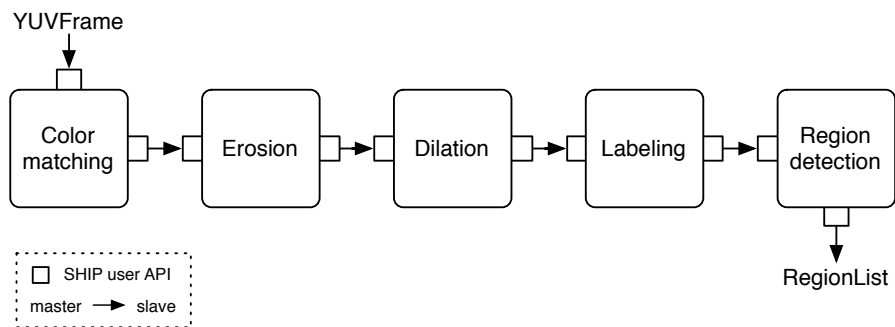


Figure C.5: Gesture recognition pipeline based on SHIP PEs

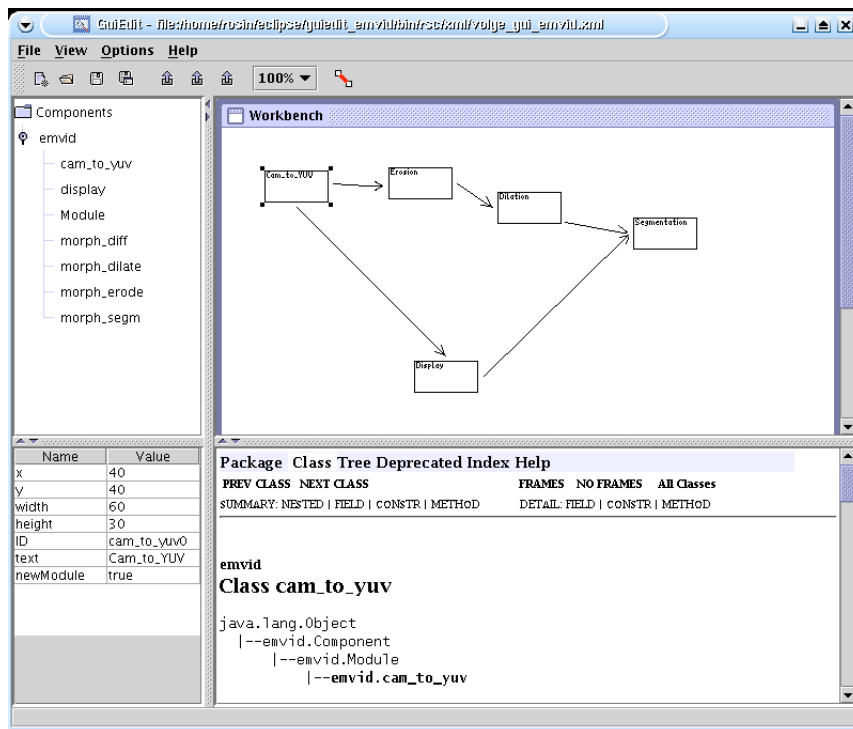


Figure C.6: Graphical composition of EmViD systems

Listing C.1: Video frame processing loop in EmViD components using SHIP

```

1 template<>
  void emvid_module_name<FRAME>::morph() {
3   FRAME frame;
   while(true) {
5     in_port.recv(frame);
     out_port.send(morph(frame));
7   }
  }

```

Listing C.1 shows the `processData` method which is uniform to all video processing components. It contains an endless loop that receives a video frame on the slave port, passes it through the component-specific `morph` algorithm, and sends the result on the master port.

C.2.2 Communication refinement for SHIP PEs

Using SHIP-based PEs as a base, a straight-forward refinement approach is to replace their PV ports with BA ports. This can be done by simple code substitution (in [72], substitution algorithms are presented in pseudo-code). The result is a model with refined ports that now use asynchronous nonblocking communication. In our approach we use OCP-tl1 as target interface (see 3.3). The computation code remains unchanged. By inserting serialization and deserialization loops, SHIP objects can be converted into OCP data streams and vice versa. To complete the refinement procedure, the designer's task is to migrate data processing into these automatically generated loops, thus altering computation from object processing to stream processing. Although automation of the computational refinement also might be possible (e.g. by using behavioral synthesis approaches such as discussed in [77]), it is not subject to this case study.

When all hardware modules have been refined to OCP, a system clock is introduced and all OCP events are replaced by synchronous wait statements. With GREENBUS the OCP PEs can interact 'mixed-mode' with SHIP PEs. To finally switch from OCP-tl1 to OCP-tl0 pin and cycle accurate RTL communication, the corresponding OCP RTL signals are introduced and the addresses used in the simulation model are converted into the target platform address space (see [72]).

Example 5. *For the EmViD gesture recognition system-on-chip, a design team may decide to implement the region detection in hardware. The corresponding PEs are therefore refined towards RTL. The refinement is performed in three steps:*

1. *The SHIP ports are replaced by OCP-tl1 ports. A memory component is attached as an explicit framebuffer on which the video processing takes place. This is necessary to eliminate the `framebuffer` class variable which cannot directly be synthesized into hardware. A BRAM (Block RAM, see [131] page 33) with TAC interface is taken from an IP library. The related code sections in the video processing components are changed to use the BRAM for data caching.*
2. *The video processing source code in the `morph` function is refined into an implicit state machine (SM) that works on the data in the BRAM. Each time the BRAM is accessed a `wait` statement is inserted. This splits the video processing into several distinct phases and adds some notion of time. With GREENBUS each refined component can be verified individually: while for all other components their SHIP PEs are used, the 'component-under-refinement' already uses the OCP user APIs for GREENBUS (fig. C.7).*
3. *The implicit SM is converted into an explicit SM, and the communication interfaces are refined to pin and cycle accurate versions (fig. C.8. The explicit SM distributes functionality over a number of parallel `SC_METHOD` processes, which are synchronized by a global state. SystemC*

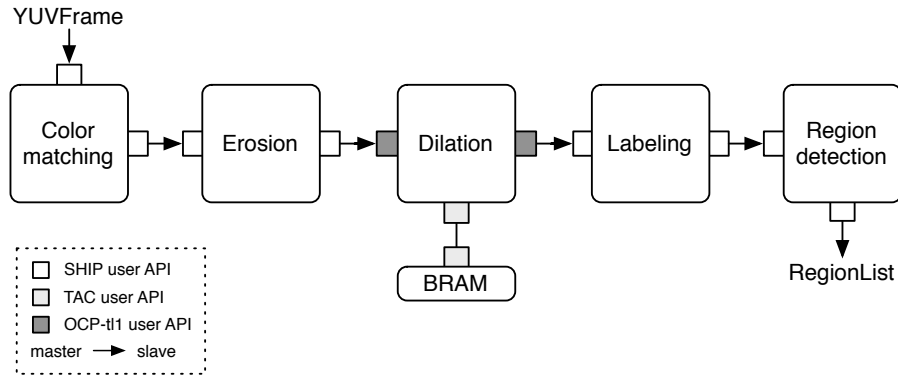


Figure C.7: The dilation component has been refined to an implicit SM with OCP-tl1 ports and attached BRAM block. The rest of the system is still at an untimed level of abstraction.

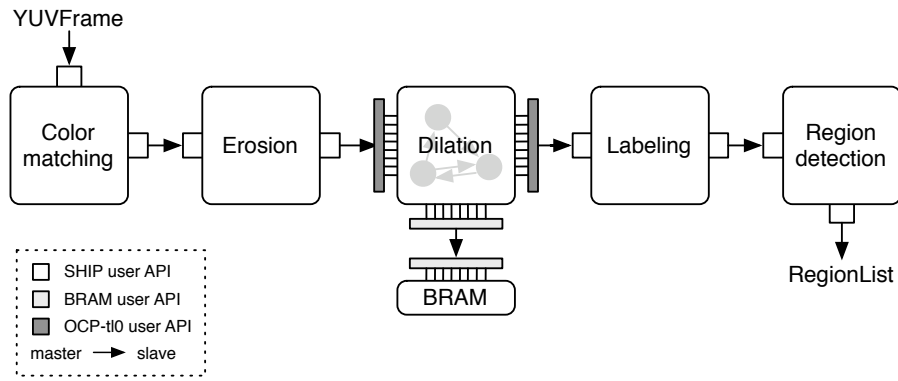


Figure C.8: The dilation component has been refined to an explicit SM with pin accurate communication.

synthesis tools are available that aim to automate this step, e.g. Celoxica Agility [20]. See [140] and [89] for an analysis of the applicability of these tools, using EmViD IPs as example.

C.2.3 Back annotation of low-level timing

As a result of top-down refinement, a less abstract model with timing information for both computation and communication is derived. These timing information can be analyzed using GREENCONTROL configurable parameters. Then, the designer can back-annotate the measured timings into the SHIP model, again using GREENCONTROL.

Example 6. *After the explicit SM version of the EmViD dilation component was created, its computation latency was surveyed. For the image size 80x60 pixels (which we decided to use for gesture recognition) a constant latency of 9961 cycles per frame was measured (fig. C.9). This value can be back-annotated into the SHIP model without changing the code, using a configurable parameter in the SHIP slave API.*

After applying back annotations to all SHIP slave ports in the EmViD golden model, it provides a computation timing accuracy close to the RTL model. The back-annotated model runs with >30 frames/sec on a standard Linux PC and thus is sufficiently fast for embedded software development.

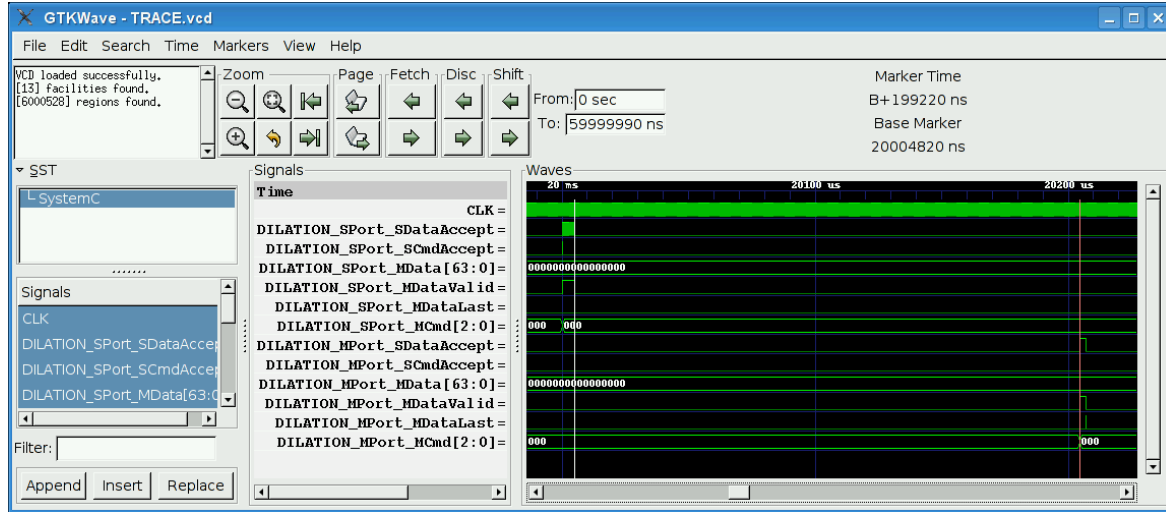


Figure C.9: Example of measuring the computation time of the refined dilation component. The waveform output was generated with GREENCONTROL. You can see two OCP transactions, one for receiving a video frame on the slave port, and one for sending the frame to the master port, after applying the dilatation algorithm. The marked section of 199220 ns is the measured delay at 100 MHz clock speed.

C.3 TRAIN and HPC

After the system refinement procedure has arrived at the point where all hardware PEs are available as RTL (including both self-implemented hardware and intellectual property), an FPGA-based prototype is ready to be generated. The TRAIN communication co-processor library presented in [70] can be used to ease the communication architecture implementation, in particular when IPs with different interfaces are used.

C.3.1 TRAIN architecture

TRAIN aims at a meet-in-the-middle approach. Each hardware core is equipped with a communication adapter called **accessor**, and each processor core connects to the on-chip communication subsystem by a **CPU adapter**. On the software side, TRAIN provides a hardware abstraction layer (HAL), which cooperates with the CPU adapter and therewith enables transactions from software processes to the accessors. The advantage of this approach is that the accessors and CPU adapters implement a platform independent protocol on top of the underlying bus protocols. Hardware and software models can be reused with different communication architectures and different processors. Roughly speaking, TRAIN accessors and CPU adapters on an FPGA are the equivalent to GREENBUS user APIs in the TLM model. An overview of TRAIN applied to a processor subsystem of a multiprocessor SoC is depicted in figure C.10. TRAIN is discussed in detail in [70].

Example 7. To connect the refined EmViD hardware PEs with OCP-tl0 interfaces to the buses on a Xilinx Virtex-II Pro FPGA [131], TRAIN accessors are used. Figure C.11 shows an example EmViD implementation using the PLB. The color matching core reads video frames from the system memory (not shown) via the PLB, thereby using a PLB accessor. It applies color matching and sends the result into the region detection pipeline. The last pipeline stage, regiondetect, uses another PLB accessor to write the list of detected regions into the system memory.

Figure C.12 depicts another conceivable architecture. Here, the pipeline stages share a single BRAM block which is connected to an OPB.

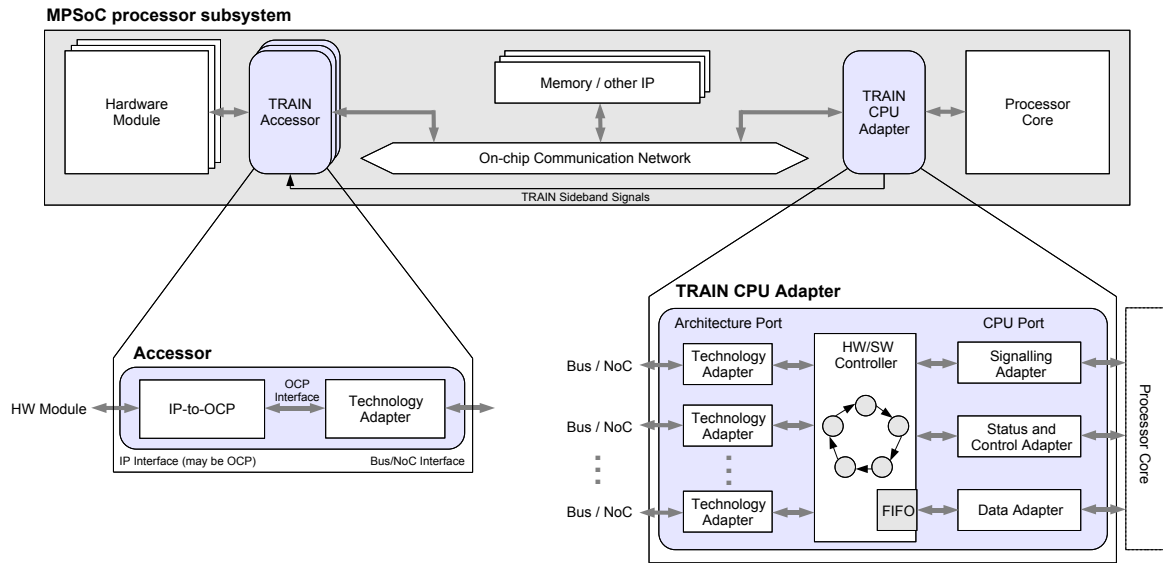


Figure C.10: TRAIN architecture overview

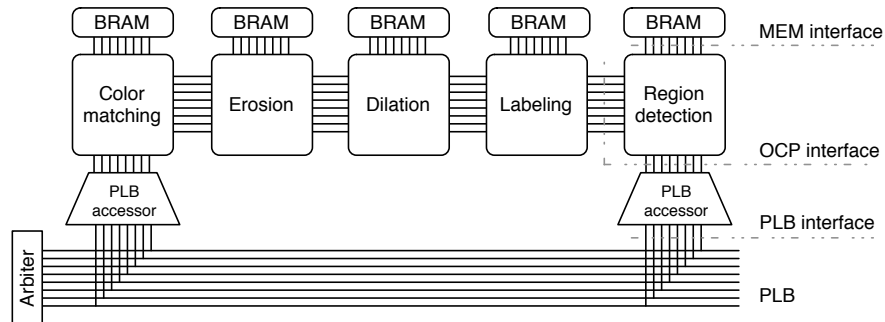


Figure C.11: EmViD cores connected to a Processor Local Bus (PLB) using accessors

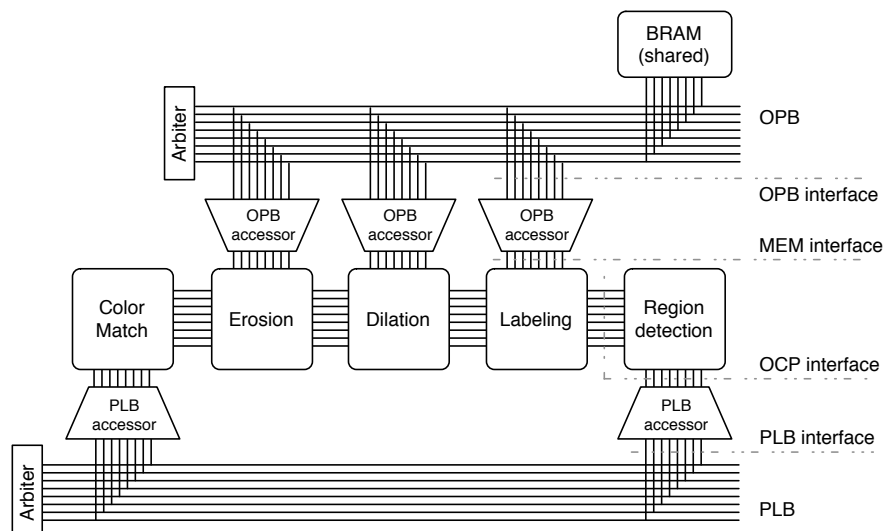


Figure C.12: EmViD architecture with a shared BRAM, using both OPB and PLB accessors

C.3.2 Tools for software development

Software PEs can also be developed using SHIP for communication both with other software PEs and with hardware PEs. A more advanced approach are hardware procedure calls (HPC): they provide hardware services as remote method calls to software PEs, such as `encodeJPEG`. In [75] a protocol for HPCs over the GREENBUS generic protocol is presented and a concept is outlined with which it may be possible in the future to generate HPC user APIs for GREENBUS automatically. In this case study, we implemented the HPC user APIs by hand.

Finally, the software PEs of the system have to be compiled for the target platform processor(s) and operating system. To this end, we adopt the methodology first presented in [48]. The GREENBUS user APIs as well as the used SystemC functions and data types are mapped to a target platform specific C/C++ library, which emulates the SystemC library. The software code that has been developed using the SystemC virtual prototype as a base therefore can be directly compiled for the target platform, without changes to the code.

Compelling as it first seems, this approach is problematic when more than one software process executes in parallel. Multi-threading in SystemC is cooperative, i.e. non-preemptive. This is in contrast to standard operating systems, which typically use preemptive scheduling and mutexes, message queues, etc. To make sure that the software behaves similar as in the simulation the SystemC process scheduler must be replicated on the target processor.

In a student research project a minimalist SystemC kernel which meets these requirements, the so-called SC- μ Kernel, has been implemented [118]. The SC- μ Kernel uses POSIX user-level context switching (see [59]) to realize the SystemC process scheduling behavior and thus works independent of the underlying operating system (if any).

C.4 Architecture exploration and synthesis of a real-time gesture recognition system

Using GREENBUS and the above described TLM design flow and tools, a complex system-on-chip for real-time recognition of body gestures has been developed. The golden model of the video processor consists of 8 PEs, namely video input, color matching, erosion, dilation, labeling, region detect, gesture recognition, and video output. This model was used for functional verification. Therefore all PEs have been implemented as abstract SystemC models, using SHIP communication over GREENBUS.

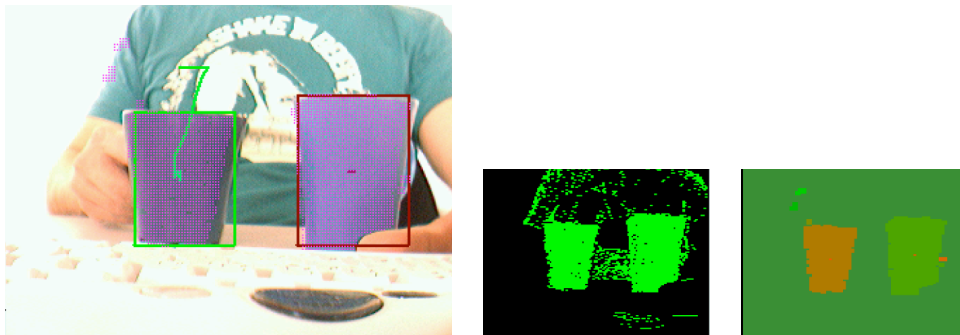


Figure C.13: Two colored cups detected by the video processor model. The left image shows the input video with superimposed detection information. The middle image shows the intermediate result after color matching, erosion, and dilation have been applied. The right image shows the detected regions after the labeling and region detection phase.



Figure C.14: Motion traces recorded by the gesture recognition PE for the detected objects (here, the two hands and the head).

Video input can provide video data from either an MPEG file, a network camera, or a FireWire camera connected to the PC. Video output uses the SDL library [82] to display the output of the video processor in a testbench window (fig. C.13).

Video processing takes place as a sequence of image manipulation steps. At first, the frames grabbed from the video source are analyzed by color matching. Pixels that are out of a given color range are set to black, all other pixels are set to white. The resulting black-and-white image is then processed by an erosion algorithm to remove noise, and dilation merges neighbored pixels to remove interference. Afterwards, the labeling phase marks all connected pixel arrays in the image, and this information is used by the region detection PE to produce a list of detected objects in the image. Finally, gesture recognition is performed by analyzing the movement of these regions over time (fig. C.14).

The video algorithms have been implemented using GREENCONTROL configurable parameters. So we were able to fine-tune them during the running simulation. When a suitable parameter configuration was found, the values were stored in a configuration file.

The golden model is depicted in figure C.15.

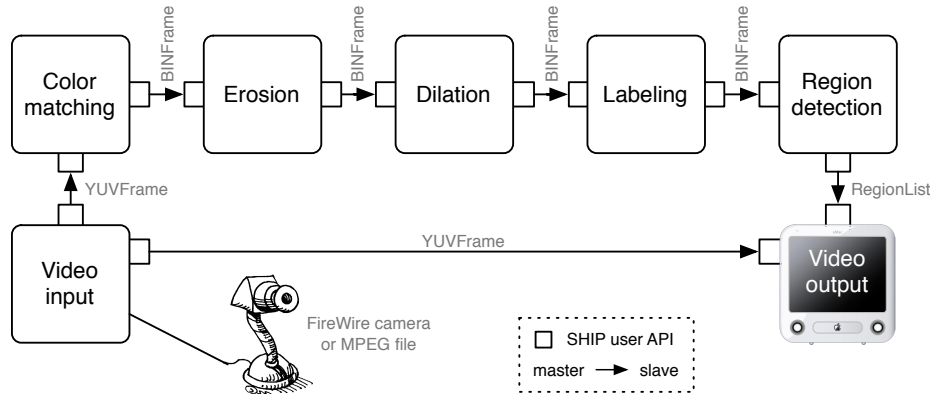


Figure C.15: Golden model of the gesture recognition system

C.4.1 Hardware/software partitioning and design flow parallelization

While the image processing algorithms are application-specific and therefore required manual refinement, other components could be taken from standard IP libraries. To start with the refinement process, the following initial design decisions have been made:

- *Hardware/software partitioning*: the functionalities of the system are partitioned into hardware and software as shown in table C.2.
- *IP reuse*: for video input and output, as well as for memories, existing IP cores are used (video digitizer, VGA display controller, BRAM, DDR RAM).

Table C.2: HW/SW partitioning of EmViD gesture recognition system-on-chip

Function	Hardware	Software
Video input	Standard digitizer IP	Init & configure core
Display	Standard VGA display controller IP	Init & configure core
Memories	Standard BRAM & DDR cores	–
Color matching	Custom-built IP	–
Erosion	Custom-built IP	–
Dilation	Custom-built IP	–
Regiondetect	Custom-built IP	Read detected regions
Recognize gestures	–	Analyze detected regions over several frames
User interface	–	Show detected gestures on the screen

Hence, the design tasks could be distributed onto three teams of engineers (cp. fig. C.1):

1. *IP team*: integration of the existing IP cores.
2. *Hardware team*: refinement of the application-specific hardware PEs for RTL synthesis.
3. *Software team*: software development using the TLM model as virtual prototype of the hardware platform.

C.4.2 IP team

The IP team decided to integrate an existing video digitizer IP core [121], a VGA display controller [33], and a DDR memory controller [133, p. 111] into the chip. For the display and the memory controllers abstract TLM models with TAC interfaces were available. The video digitizer was only provided as gate netlist with OCP interface. Figure C.16 shows the EmViD virtual prototype after bottom-up integration of these cores. Note the mix of different user APIs.

Note also that a GREENBUS router is introduced, as both the display controller and the video digitizer work on the DDR memory, so that a shared memory bus is required. However, as no decision about the target communication architecture has been made, a dummy protocol class is used that does not implement any timings or scheduling schemes.

C.4.3 Hardware team

The hardware team had two tasks. First, it refined the application-specific hardware PEs towards an RTL representation that can be fed into a hardware synthesis tool (we used Celoxica Agility [20]). Each PE was refined individually, using the step-by-step approach described in C.2.2. The outcome were explicit SM RTL models of the PEs with OCP interfaces.

Secondly, GREENBUS was used to identify a suitable communication architecture. While more and more refined hardware cores became available, their timing information was back-annotated to the golden model (see C.2.3).

We experimented with different communication architecture configurations for the design. As one might expect, some architectures are better suited than others to meet efficiency requirements such as a given frame rate.

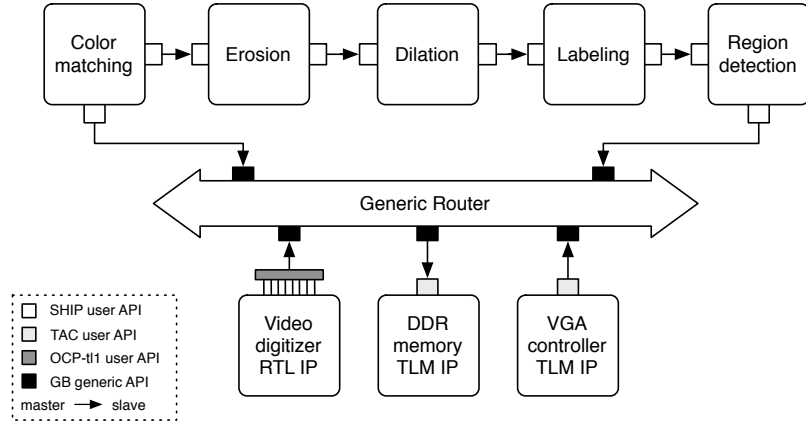


Figure C.16: The IP cores video digitizer, display, and DDR memory have been integrated into the TLM model bottom-up.

C.4.3.1 Pipeline architecture

In figure C.17 the gesture recognition subsystem is organized as a pipeline. The target platform is a PowerPC-based system-on-chip, hence the main memory is connected to a PLB. Video input is connected to the PLB as master and writes the grabbed video data to a DDR RAM memory area. The first pipeline component is connected to an OPB, which allows DDR RAM access via an OPB-to-PLB bridge. It reads a video frame from the DDR RAM, processes it (color matching), and writes the result to a dedicated RAM block (BRAM) on the FPGA. The frame in the BRAM is processed by the subsequent pipeline component, which writes its result in another BRAM, and so forth. Finally, the pipeline output is written to another DDR RAM memory region, from where it can be read by the software.

The pipeline architecture assures reliable and fluent video processing. All PEs work in parallel, such that color matching can already process the second frame while erosion processes the first frame. Figure C.18 illustrates this behavior. It can be seen that after the pipeline has been ‘filled’, the pipeline outputs one frame every two clock cycles¹.

¹For the sake of simplicity we here assume that processing is done in zero time, i.e. computation is modeled untimed.

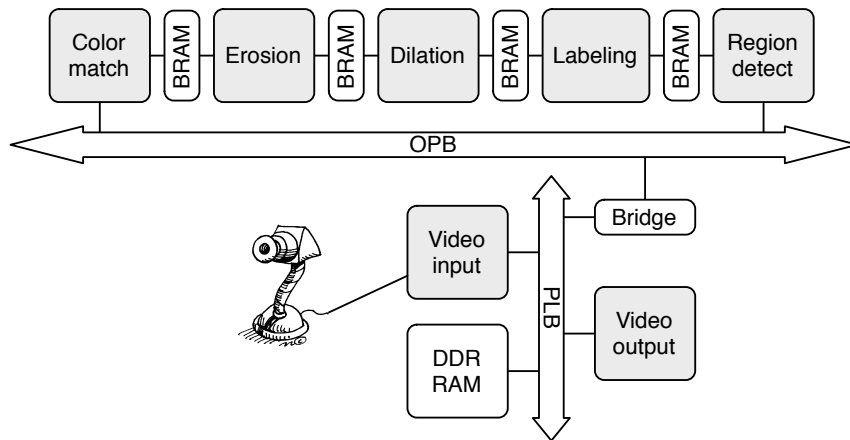


Figure C.17: EmViD pipeline architecture

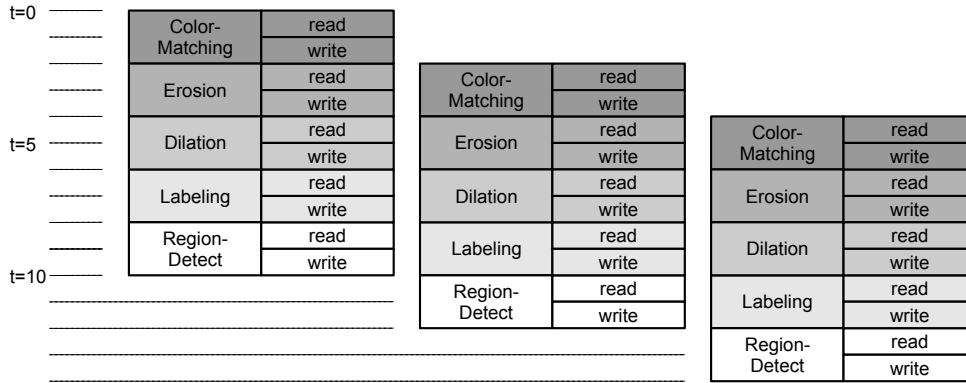


Figure C.18: Data flow in the pipeline architecture

Listing C.2: Configuration file for the EmViD pipeline architecture

```

# Buses
2  BUS(plb, PLB, 40, SC_NS)
  BUS(opb, OPB, 40, SC_NS)
4
# PLB subsystem
6  CONNECT(video_input.iport, plb.tport)
  CONNECT(plb.iport, ddr_ram.tport)
8  CONNECT(video_output.iport, plb.tport)
10
# Pipeline
  CONNECT(color_matching.in, opb.tport)
12 CONNECT(color_matching.out, bram1.port1)
  CONNECT(erosion.in, bram1.port2)
14 CONNECT(erosion.out, bram2.port1)
  CONNECT(dilation.in, bram2.port2)
16 [...]
  CONNECT(labeling.out, opb.tport)
18
# Bus bridge
20 CONNECT(opb.iport, bridge.tport)
  CONNECT(bridge.iport, plb.tport)

```

The pipeline architecture is set up with the platform configuration file in listing C.2 (using GREEN-CONTROL). GREENBUS then exactly simulates the communication architecture shown in figure C.17. The simulation results match the timing shown in figure C.18.

C.4.3.2 Bus architecture

The high efficiency of the pipeline architecture is bought with a lot of BRAM; using 2 byte per pixel at 320x240 pixels per frame each BRAM block allocates 150 KByte on-chip memory. This is too much for our target platform, the Virtex-II Pro V2P30 FPGA, which only provides 306 KByte of on-chip memory in total.

As an alternative solution, consider the bus architecture in figure C.19. In this model, all pipeline stages have been connected to the OPB. Instead of dedicated BRAM blocks they use the shared system memory (DDR RAM) to store the intermediate results. Only one read or write transaction can take place at a time. As can be seen in figure C.20 this affects the timing behavior of the system significantly.

All pipeline stages are courting the bus arbiter simultaneously. Thus, great attention must be paid to the scheduling scheme. For the OPB, a fixed priority scheduler in combination with ascending

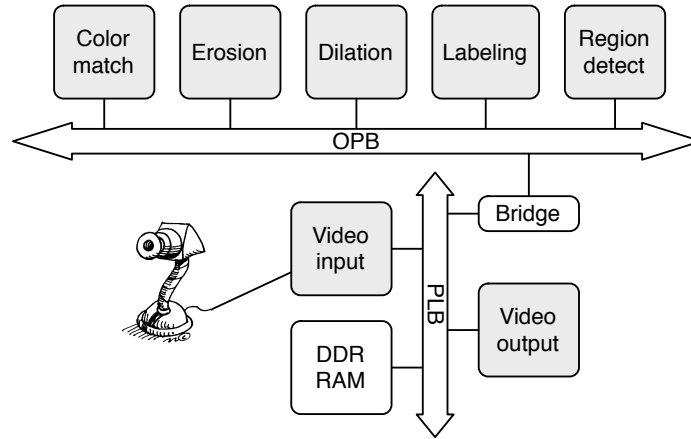


Figure C.19: EmViD bus architecture

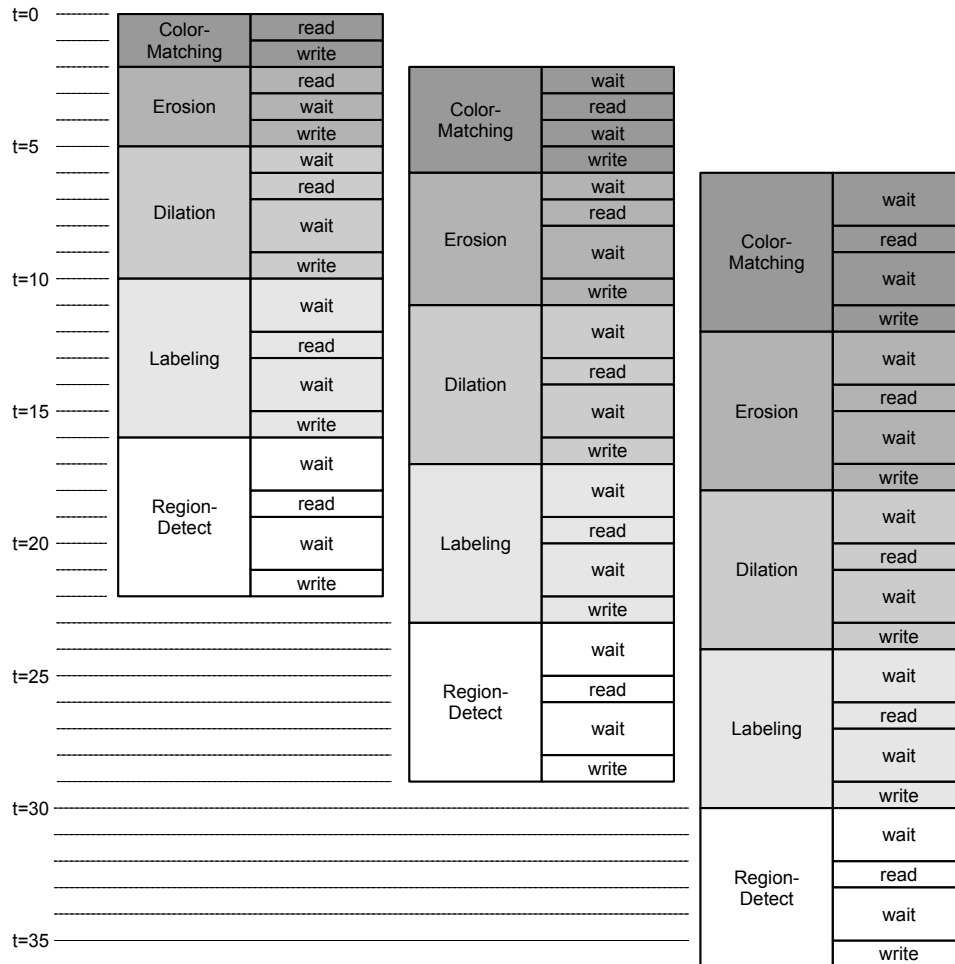


Figure C.20: Data flow in the bus architecture using fixed ascending priorities

priorities for the pipelined stages turned out to be a good solution. For the PLB dynamic priority scheduling is used. With repeated GREENBUS simulation, deadlock or starvation situations due to inappropriate scheduling schemes were identified.

C.4.3.3 Communication analysis

We utilized configurable parameters to count the number of executions for the various processes in the model in order to identify the location of communication bottlenecks in design configurations with poor frame rates. Table C.3 presents some results of the experiments².

Table C.3: EmViD execution traces

Architecture	# exe. video	# exe. detect.	FPS video	FPS detect.	Comment
Bus only model 1	500	500	25	25	ascending priority
Bus only model 2	451	872	22.55	43.35	higher detection priority
Mixed bus/pipeline model 1	500	1000	25	25	lower pipeline priority
Mixed bus/pipeline model 2	500	1000	25	50	higher pipeline priority

The column ‘#ex video’ shows the total number of video frames successfully sent from video input to video output. The column ‘#ex detect.’ shows the total number of video frames processed by the last pipeline stage (regiondetect). From these numbers overall frame rates have been calculated (columns ‘FPS video’ and ‘FPS detect.’).

Row 1 and row 2 show the frame rates we got with the bus architecture model. While in row 1, the bus access priorities were assigned in ascending order according to the sequence of video processing stages in the model, in row 2 I assigned a higher priority to the region detection components than to the video display data path. As expected, the frames per second processed for region detection goes up, but as an unintentional side effect due to higher bus workload, the number of video frames displayed per second drops down.

Rows 3 and 4 show the results I achieved with the pipeline architecture model. Here, I could considerably increase the video display frame rate by just swapping the bus access priorities of two components. With this set up, 25 frames per second full resolution live video display is achieved while the region detection runs at the high rate of 50 frames per second.

C.4.4 Software team

The gesture recognition is realized in software. By tracking the regions detected by the hardware over a number of frames, predefined body gestures can be recognized. The software therefore considers size, placement, and motion of the detected regions.

The software was developed in SystemC using the bottom-up integration model (fig. C.16) as simulation platform. Hardware procedure calls on top of GREENBUS (see [75] for a detailed description of this approach) are used to initialize and configure the video input and display IP cores. The TAC user API is used to read the regiondetect output from the memory. A library of drawing functions was implemented (`setPixel`, `drawRect`, etc.) that use TAC interface method calls to modify pixels of the framebuffer in main memory. With these, the software marks the detected regions and gestures in the live video.

To give an example, listing C.3 shows the `setPixel` method. It converts the given RGB value into the YUV color space and writes the value to the frame buffer. The TAC interface method call takes place in line 9. Note that there is no difference between the SystemC model and the software code on the final system, thanks to TRAIN (see C.3).

²These experimental results have also been presented in [43].

Listing C.3: The setPixel method uses TAC for HW/SW communication

```

1 void sw::setPixel(uint x, uint y, uint r, uint g, uint b) {
    std::vector<gs_uint8> p(4);
3
    p[0] = (gs_uint8)(0.257*(float)r + 0.504*(float)g + 0.098*(float)b + 16);
5    p[1] = (gs_uint8)(-0.148*(float)r - 0.291*(float)g + 0.439*(float)b + 128);
    p[2] = p[0];
7    p[3] = (gs_uint8)(0.439*(float)r - 0.368*(float)g - 0.071*(float)b + 128);
9
    mem_port.write(p, VIDEO_MEMORY_ADDRESS_BUF2+(y<<12)+x*2, 4);
}

```

C.4.5 Integration and verification

Finally, the products of the three design teams were integrated in an overall integration model of the system. While the intermediate models that were produced during the refinement procedure simulate indeed slower than the golden model but still reach a performance good enough for iterative ‘refinement-verify’ cycles, the final integration model is exceedingly slow. All hardware cores reside at the RTL level of abstraction and therefore the simulator needs to consider every single clock edge. However, due to their blocking user APIs, synchronization of the faster software models via GREENBUS is unproblematic.

Table C.4 gives an overview on the simulation performance of different models of the hardware/-software system. The measurements show the simulation durations for processing 1,000 frames of a test video on an Athlon XP 2600+ system.

Table C.4: Simulation performance of EmViD models at different abstraction levels

Model	Untimed	Implicit SM	Explicit SM	Pin accurate	RTL
CPU time [s]	0.028	7,516.7	11,640.8	14,608.9	16,671.1

Figure C.21 shows timing estimations made with these models at the different stages of the design cycle. The measurement shows the number of required computation and communication clock cycles for region detection as they are predicted per frame by the four timed models. The used test stimuli is a video sequence from the famous Star Trek TV series. It contains a changing number of actor heads and hands that are detected by the models. More actors in the picture lead to more detection and communication overhead.

It can be seen that both the pin accurate and the RTL models produce the same results. In contrast to the pin accurate model which has been written by hand (and uses TLM communication), the RTL model was generated out of it with Celoxica’s Agility compiler. The timing estimation of the more abstract explicit SM model also comes very close to the RTL simulation. But even the early implicit SM model produces a reasonable prognosis. It contains just the `wait` statements that implicitly indicate the state changes. The communication times are already precisely simulated by GREENBUS, using the BA level of abstraction. It can be seen in the graph that this combination delivers a good estimation. In comparison to the results achieved with the other models, there is a constant miss of 9930 clock cycles per frame. The reason is that the BRAM accesses are still modeled at a functional level and therefore are considered ‘untimed’.

C.4.6 System synthesis

We implemented a hardware abstraction layer for the PowerPC 405 processor which is immersed in Xilinx Virtex-II Pro FPGAs, so that the software can be executed on the FPGA. Hardware-software communication is established with a TRAIN CPU adapter (see C.3). To connect the hardware cores to the PLB and OPB buses on the FPGA, appropriate TRAIN accessors have been developed.

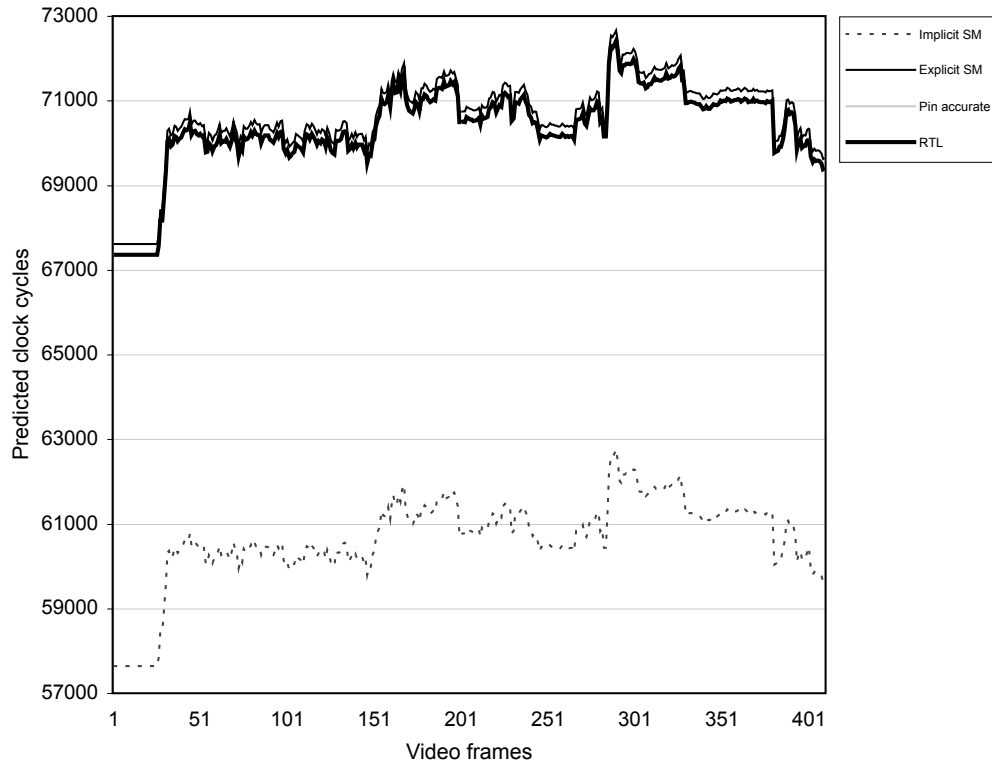


Figure C.21: The graph shows the predicted number of clock cycles per frame for region detection, using the differently abstract TLM models

Xilinx Embedded Development Kit v8.2 [136] was used to compile the complete hardware/software system for the XUP prototyping board [134]. Table C.5 lists the FPGA resource allocation of the EmViD components. Figure C.22 shows the FPGA floor plan after place-and-route. The slices allocated by the different components are distinguished with different colors. The two boxes are the PowerPCs. The largest area is allocated by the labeling and region detection hardware.

Table C.5: Virtex-II Pro FPGA resource allocation of the EmViD components

Component	Slices	FFs	4-LUTs	IOs	Special resources
Dilation	1,147	807	2,142	550	
Erosion	1,062	835	2,013	550	
Labeling and Region	7,608	1,297	14,352	552	
Video Input	432	589	758	295	16 BRAMs
Video Output	512	639	816	265	10 MULT18x18s
BRAM blocks	0	0	0	1,284	67 BRAMs
PLB master accessor	190	310	300	630	
PLB slave accessor	176	272	215	426	
Software	0	0	0	855	1 PowerPC-405

The FPGA implementation of our system runs at 100 MHz (both the hardware cores and the PowerPC processor) and easily processes the 30 frames per second that are grabbed from the connected video camera. We repeated the timing measurement from above on the FPGA. The FPGA measurement precisely confirmed the simulation results we got with GREENBUS.

Finally, figure C.23 shows a photo of our testbench. On the left-hand display you can see the SystemC simulation of the hardware/software platform, running on a Linux PC with a FireWire camera for video input. On the right-hand monitor you can see the video output of the FPGA, which

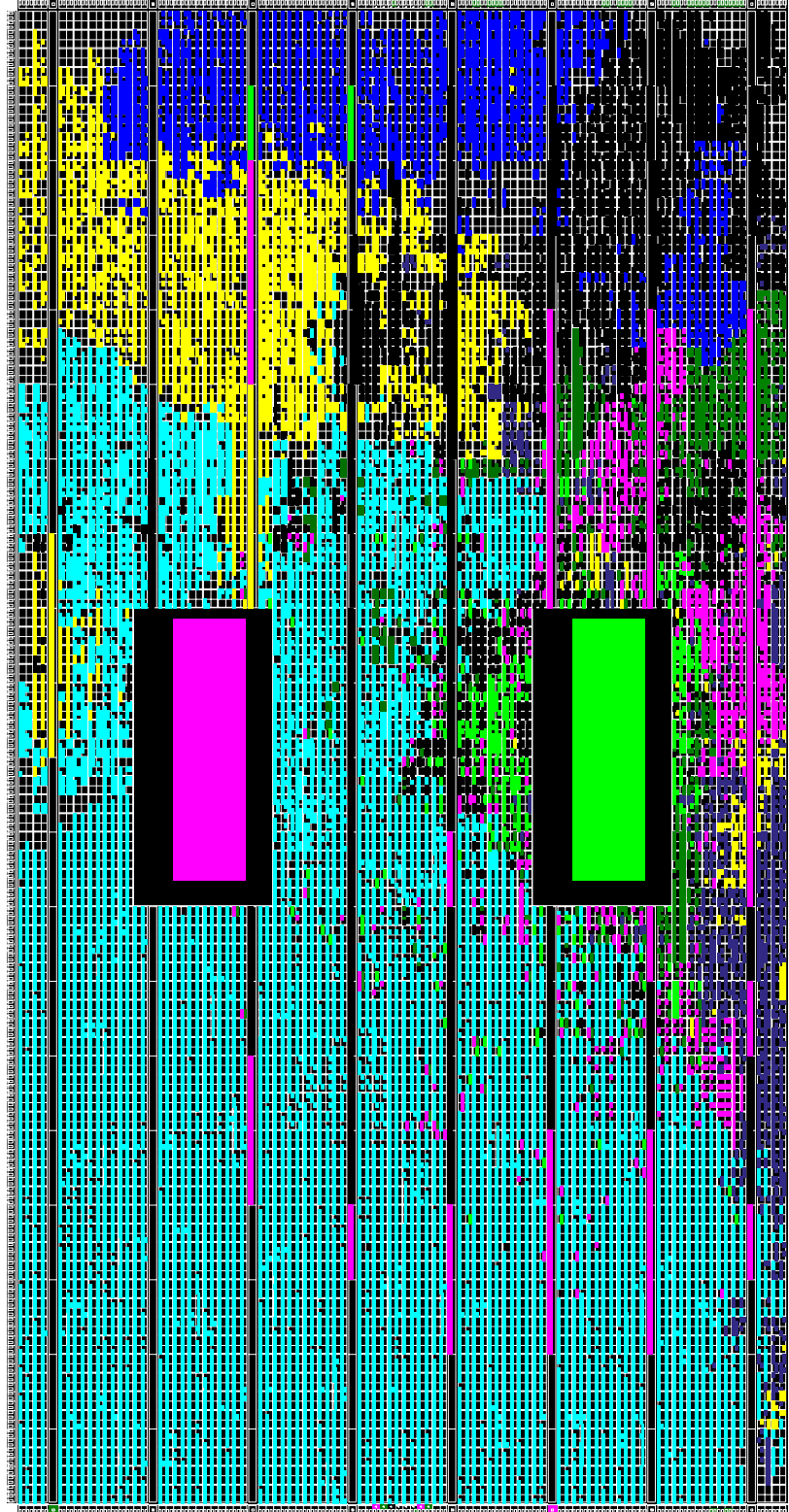


Figure C.22: EmViD SoC floorplan for the Virtex-II Pro FPGA V2P30

has been programmed with our system-on-chip (the XUP prototyping board with the FPGA can be seen on the table in front of the monitor). A PAL camera has been connected to the FPGA. It can be seen that the SystemC simulation produces the same video output as the FPGA. Furthermore, you can see our heads and hands being detected by both the virtual and the real system implementation.



Figure C.23: Our EmViD testbench in the lab

C.5 Conclusion and outlook

This case study has shown GREENBUS to be appropriate for communication modeling in all phases of a TLM design flow, starting with high-level untimed communication in the golden model and reaching over various intermediate mixed-mode models to the final cycle accurate implementation model. Experimenting with different architecture variants becomes exceedingly easy with GREENBUS, as it enables the seamless integration of heterogeneous PEs to a new model by simply writing a new configuration file.

The comparison of our measurements on the FPGA with the measurements made with GREENBUS shows that the bus accurate abstraction approach can provide very precise simulation results in terms of the timing of the analyzed communication architecture. However, the accuracy strongly depends on the careful estimation of the computation timings in the PEs as well. The achieved simulation performance confirms the expectation in respect of the abstraction of the connected PEs, although the results of the case study also show that the high simulation performance of GREENBUS CAFMs quickly loses ground when the computation simulation in the PEs is generally slow. Using back-annotation, still very high simulation speeds can be achieved.

Using GREENBUS as a base, an outlook on further TLM design tools (TRAIN, HPC) has been given which pick up on the GREENBUS idea of decoupling model interfaces and target platform interfaces in order to enable better design reuse and inter-operation. This opens up a large field of further research challenges towards raising the abstraction level required for automatic hardware/software synthesis from RTL to TLM.

Bibliography

- [1] ACTIS DESIGN. *AccurateC: Static C++ Code Analysis for SystemC*. Datasheet, Actis Design, Portland, USA, 2005.
- [2] ADLER, D., COLVIN, G., AND DAWES, B. *Boost library smart pointer documentation: http://www.boost.org/libs/smart_ptr/smart_ptr.htm (last checked: 02/12/08)*. Boost Developers Group, 2005.
- [3] ALDIS, J. *Platform Based Design at the Electronic System Level*. Springer, Dordrecht, Netherlands, 2006, ch. Use of SystemC modelling in creation and use of an SoC platform: Experiences and lessons learnt from OMAP-2, pp. 31–47.
- [4] APACHE SOFTWARE FOUNDATION. *XMLBeans homepage: <http://xmlbeans.apache.org> (checked 02/12/08)*. The Apache Software Foundation, Forest Hill, USA, 2007.
- [5] ARM. *AMBA Specification (Rev 2.0)*. Datasheet, ARM Ltd., USA, 1999.
- [6] ARM. *AMBA 3 APB Protocol v1.0 Specification*. Datasheet, ARM Ltd., USA, 2004.
- [7] ARM. *AMBA 3 AXI Protocol v1.0 Specification*. Datasheet, ARM Ltd., USA, 2004.
- [8] ARM. *AMBA 3 AHB-Lite Protocol v1.0 Specification*. ARM Ltd., USA, 2006.
- [9] ARM. *RealView ESL API, available at: <http://www.arm.com/products/DevTools/RealViewESLAPIs.html> (checked 02/12/08)*. ARM Ltd., USA, 2007.
- [10] BERGAMASCHI, R. Transaction-Level Models for PowerPC and CoreConnect. In *Proc. 11th European SystemC Users Group Meeting* (Munich, March 2005).
- [11] BERRADA, M., AND ALDIS, J. SystemPython: a Python extension to control SystemC SoC simulations. In *Proc. Design and Test in Europe Conference (DATE)* (Nice, April 2007).
- [12] BJERREGAARD, T., AND MAHADEVAN, S. A survey of research and practices of Network-on-Chip. *ACM Computing Surveys (CSUR)* 38, 1 (New York, USA, 2006).
- [13] BJERREGAARD, T., MAHADEVAN, S., OLSEN, R. G., AND SPARSO, J. An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip. In *Proc. Int. Symposium on System-on-Chip* (Tampere, Finland, November 2005).
- [14] BLACK, D. C., AND DONOVAN, J. *SystemC From the Ground Up*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2004.
- [15] BODE, C. Ein Java-Framework zur Verwaltung von SystemC-Simulationsdaten in SQL-Datenbanken. Diploma thesis, TU Braunschweig, Abt. E.I.S., April 2007.
- [16] BURTON, M., ALDIS, J., GÜNZEL, R., AND KLINGAUF, W. Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified. In *Proc. Forum on Design Languages (FDL)* (Barcelona, September 2007).
- [17] BURTON, M., AND MORAWIEC, A., Eds. *Platform Based Design at the Electronic System Level: Industry Perspectives and Experiences*. Springer, Dordrecht, Netherlands, 2006.

- [18] CAI, L., AND GAJSKI, D. Transaction Level Modeling: An Overview. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Newport Beach, USA, October 2003).
- [19] CALDARI, M., CONTI, M., COPPOLA, M., CURABA, S., PIERALISI, L., AND TURCHETTI, C. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, 2003).
- [20] CELOXICA. *Agility Manual for Agility 1.1*. Celoxica Ltd., Oxfordshire, UK, 2006.
- [21] CESARIO, W., BAGHDADI, A., GAUTHIER, L., LYONNARD, D., NICOLESCU, G., PAVIOT, Y., YOO, S., JERRAYA, A. A., AND DIAZ-NAVA, M. Component-based design approach for multicore socs. In *Proc. Design Automation Conference (DAC)* (New Orleans, USA, June 2002).
- [22] CESARIO, W., AND JERRAYA, A. *Multiprocessor System-on-Chip*. Morgan Kaufmann Publishers, San Francisco, USA, 2004, ch. Component-Based Design for Multiprocessor Systems-on-Chip, pp. 357–393.
- [23] CHANG, H., COOKE, L., HUNT, M., MARTIN, G., MCNELLY, A., AND TODD, L. *Surviving the SOC Revolution*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1999.
- [24] CHEVALIER, J., DE NANCLAS, M., FILION, L., BENNY, O., RONDONNEAU, M., BOIS, G., AND M. ABOULHAMID, E. A SystemC Refinement Methodology for Embedded Software. *IEEE Design & Test of Computers* 23, 2 (New York, March-April 2006), 148–158.
- [25] COPPOLA, M., CURABA, S., GRAMMATIKAKIS, M., MARUCCIA, G., AND PAPARIELLO, F. *On-Chip Communication Network: User Manual V1.0.1*. ST Microelectronics, France, and ISD S.A., Greece, October 2003.
- [26] CoWARE. *Platform Architect - SystemC Platform Capture and Analysis for Platform-driven ESL Design*, available at <http://coware.com/products/platformarchitect.php> (checked 02/12/08). CoWare Inc., San Jose, USA, 2006.
- [27] CoWARE. *CoWare Model Library*, available at <http://coware.com/products/modellibrary.php> (checked 02/12/08). CoWare Inc., San Jose, USA, 2007.
- [28] CUMMINGS, C. E. Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill! In *Proc. Synopsys User Group Conference (SNUG)* (San Jose, USA, October 2000).
- [29] DAVARE, A., DENSMORE, D., MEYEROWITZ, T., PINTO, A., SANGIOVANNI-VINCENTELLI, A., YANG, G., ZENG, H., AND ZHU, Q. A Next-Generation Design Framework for Platform-Based Design. In *Proc. Conference on Using Hardware Design and Verification Languages (DVCon)* (San Jose, USA, February 2007).
- [30] DENSMORE, D., DONLIN, A., AND SANGIOVANNI-VINCENTELLI, A. *Platform Based Design at the Electronic System Level*. Springer, Dordrecht, Netherlands, 2006, ch. Programmable Platform Characterization for System Level Performance Analysis.
- [31] DHANWADA, N., BERGAMASCHI, R., DUNGAN, W., NAIR, I., GRAMANN, P., DOUGHERTY, W., AND LIN, I.-C. Transaction-level modeling for architectural and power analysis of PowerPC and CoreConnect-based systems. *Design Automation for Embedded Systems* 10, 2-3 (Springer, Norwell, USA, September 2005), 105–125.
- [32] DONLIN, A. Transaction Level Modeling: Flows and Use Models. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Stockholm, 2004).

-
- [33] FLÄMIG, T. Entwurf eines optimierten Videocontrollers für das XUP Board. Studienarbeit, Abt. E.I.S., TU Braunschweig, March 2007.
 - [34] GAJSKI, D., ZHU, J., AND DÖMER, R. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.
 - [35] GAUTHIER, L., YOO, S., AND JERRAYA, A. A. Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2001).
 - [36] GEFFKEN, M. Analyse und Visualisierung von SystemC-Simulationsdaten durch Reflection und Introspection mit XML. Diploma thesis, TU Braunschweig, Abt. E.I.S., Januar 2006.
 - [37] GERIN, P., SHEN, H., CHUREAU, A., BOUCHHIMA, A., AND JERRAYA, A. A. Flexible and executable hardware/software interface modeling for multiprocessor soc design using systemc. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)* (Yokohama, Japan, January 2007).
 - [38] GERSTLAUER, A. Communication Abstractions for System-Level Design and Synthesis. Tech. Rep. CECS-03-30, Center for Embedded Computer Systems, University of California, Irvine, USA, October 2003.
 - [39] GERSTLAUER, A., SHIN, D., DÖMER, R., AND GAJSKI, D. System-Level Communication Modeling for Network-on-Chip Synthesis. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)* (Shanghai, China, January 2005).
 - [40] GERSTLAUER, A., YU, H., AND GAJSKI, D. RTOS Modeling for System Level Design. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2003).
 - [41] GHENASSIA, F. *Transaction-Level Modeling with SystemC*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2006.
 - [42] GRABBE, C., GRÜTTNER, K., AND SCHUBERT, T. Specification of Hardware/Software Communication Design Methodology based on Abstract Communication Models. Tech. Rep. ICODES/OFFIS/R/D11/1.1, Universität Oldenburg, Abt. Eingebettete Hardware/Software-Systeme, 2005.
 - [43] GROSSE, D., PERAZA, H., KLINGAUF, W., AND DRECHSLER, R. Measuring the Quality of a SystemC Testbench by using Code Coverage Techniques. In *Proc. Forum on Design Languages (FDL)* (Barcelona, September 2007).
 - [44] GRÖTKER, T., LIAO, S., MARTIN, G., AND SWAN, S. *System Design with SystemC*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.
 - [45] GÜNZEL, R. Rapid Prototyping von transaktionsbasiert modellierten Hardware-Software-Systemen mit SystemC. Diploma thesis, TU Braunschweig, Abt. E.I.S., April 2005.
 - [46] GÜNZEL, R., AND KLINGAUF, W. Towards a natively NoC capable OCP implementation. Tech. rep., TU Braunschweig, Abt. E.I.S., and GreenSocs Ltd., May 2007.
 - [47] GÜNZEL, R., KLINGAUF, W., AND ALDIS, J. Combinatorial Dependencies in Transaction Level Models. In *Proc. Forum on Design Languages (FDL)* (Barcelona, September 2007).
 - [48] HERRERA, F., POSADAS, H., SANCHEZ, P., AND VILLAR, E. Systematic Embedded Software Generation from SystemC. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2003).

- [49] HERVEILLE, R. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Available at: <http://www.opencores.org/projects.cgi/web/wishbone/wbspec.b3.pdf> (checked 02/12/08). OpenCores Organization, 2002.
- [50] HEUER, T. Realisierung eines JPEG-Encoders auf dem ML310. Study thesis, TU Braunschweig, Abt. E.I.S., July 2007.
- [51] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice Hall International, New Jersey, USA, 1985.
- [52] HOFER, F. High-Level- und RTL-Design eines JPEG-Encoders mit SystemC. Study thesis, TU Braunschweig, Abt. E.I.S., April 2004.
- [53] IBM. *The CoreConnect Bus Architecture*. IBM Corp., New York, USA, 1999.
- [54] IBM. *32-Bit Device Control Register Bus*. IBM Corp., New York, USA, 2000.
- [55] IBM. *On-Chip Peripheral Bus Architecture Specifications, Version 2.1*. IBM Corp., New York, USA, 2001.
- [56] IBM. *128-bit Processor Local Bus Architecture Specifications*. IBM Corp., New York, USA, 2004.
- [57] IBM. *IBM PowerPC 405 Evaluation Kit with CoreConnect SystemC TLMs*. IBM Corp., New York, USA, September 2005.
- [58] IEEE COMPUTER SOCIETY. *IEEE Standard 802: Local and Metropolitan Area Networks: Overview and Architecture*. Available at <http://standards.ieee.org/getieee802/802.html> (checked 02/12/08). The Institute of Electrical and Electronics Engineers, Inc., New York, USA, March 2002.
- [59] IEEE COMPUTER SOCIETY. *IEEE Standard 1003.1: Portable Operating System Interface (POSIX)*. The Institute of Electrical and Electronics Engineers, Inc., New York, USA, 2003.
- [60] IEEE COMPUTER SOCIETY. *IEEE Standard 1666: SystemC Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., New York, USA, 2006.
- [61] JANSSEN, M., HILDERINK, R., AND KEDING, H. Simple Version of an Abstract Bus Model. *Part of the SystemC 2.0 package*, available at <http://www.systemc.org/downloads/standards> (checked 02/12/08) (January 2002).
- [62] JANTSCH, A., AND SANDER, I. Models of computation and languages for embedded system design. In *IEEE Proceedings on Computers and Digital Techniques* (New York, March 2005), vol. 152, pp. 114–129.
- [63] JANTSCH, A., AND TENHUNEN, H. *Networks on Chip*. Kluwer Academic Publishers, Boston/-Dordrecht/London, 2003, ch. Will Networks on Chip close the Productivity Gap?
- [64] JANTSCH, A., AND TENHUNEN, H., Eds. *Networks on Chip*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2003.
- [65] KAHN, G. The semantics of a simple language for parallel programming. *Technical Report, IRIA Laboria, Rocquencourt, France and Commissariat a l'Energie Atomique, France* (1974).
- [66] KEUTZER, K., MALIK, S., NEWTON, A., RABAEY, J., AND SANGIOVANNI-VINCENTELLI, A. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 12 (New York, December 2000), 1523–1543.

-
- [67] KLINGAUF, W. GreenBus - Concepts and Implementation. Tech. rep., Abt. E.I.S., TU Braunschweig, February 2008.
- [68] KLINGAUF, W. *GreenBus Homepage*: <http://www.greensocs.com/GreenBus> (checked 02/12/08). GreenSocs Ltd., Cambridge, UK, 2008.
- [69] KLINGAUF, W. Systematic Transaction Level Modeling of Embedded Systems with SystemC. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2005), pp. 566–567.
- [70] KLINGAUF, W., GÄDKE, H., AND GÜNZEL, R. TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of synthesizable MPSoC. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2006), pp. 1318–1323.
- [71] KLINGAUF, W., AND GEFFKEN, M. Design Structure Analysis and Transaction Recording in SystemC Designs: A Minimal-Intrusive Approach. In *Proc. Forum on Design Languages (FDL)* (Darmstadt, September 2006).
- [72] KLINGAUF, W., AND GÜNZEL, R. Rapid Prototyping with SystemC and Transaction Level Modeling. Tech. rep., TU Braunschweig, Abt. E.I.S., May 2005.
- [73] KLINGAUF, W., AND GÜNZEL, R. From TLM to FPGA: Rapid Prototyping with SystemC and Transaction Level Modeling. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)* (Singapore, December 2005), pp. 285–286.
- [74] KLINGAUF, W., GÜNZEL, R., BRINGMANN, O., PARFUNTSEU, P., AND BURTON, M. GreenBus - A Generic Interconnect Fabric for Transaction Level Modelling. In *Proc. Design Automation Conference (DAC)* (San Francisco, USA, July 2006), pp. 905–910.
- [75] KLINGAUF, W., GÜNZEL, R., AND SCHRÖDER, C. Embedded Software Development on top of Transaction-Level Models. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Salzburg, October 2007).
- [76] KNORR, W. Kopplung abstrakter SystemC-Modelle von Infotainment-Komponenten mit einer Integrationstestumgebung. Diploma thesis, TU Braunschweig, Abt. E.I.S., July 2007.
- [77] KOCH, A. *Advances in Adaptive Computer Technology*. Habilitation, Technical University Braunschweig, 2004.
- [78] KOGEL, T., DOERPER, M., WIEFERINK, A., LEUPERS, R., ASCHEID, G., AND MEYR, H. A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks. *Proc. CODES+ISSS* (2003).
- [79] KOGEL, T., AND HAVERINEN, A. OCP TLM for Architectural Modeling. Tech. rep., Coware Inc., San Jose, USA, 2004.
- [80] KÖNIG, H. *Protocol Engineering*. Teubner Verlag, Munich, 2003.
- [81] KRAUSE, M., BRINGMANN, O., AND ROSENSTIEL, W. Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems. *Design Automation for Embedded Systems 10*, 4 (Springer, Norwell, USA, January 2005), 229–251.
- [82] LANTINGA, S. *Simple Directmedia Layer (SDL) homepage*: <http://www.libsdl.org> (checked 02/12/08). Sam Lantinga, 2008.
- [83] LE MOIGNE, R., PASQUIER, O., AND CALVEZ, J.-P. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Paris, February 2004), pp. 82–87.

- [84] LEE, E. A., AND MESSERSCHMITT, D. G. Synchronous Data Flow. In *Proc. of the IEEE* (September 1987), vol. 75, pp. 1235–1245.
- [85] MCGRATH, D. EE Times article: Report says EDA market could be nearing ‘inflection point’. *EETimes*, available at <http://eetimes.com/news/design/business/showArticle.jhtml?articleID=175701340> (checked 02/12/08) (December 2005).
- [86] MENTOR GRAPHICS. *Vista SystemC IDE and Debug Platform*. Available at: http://www.mentor.com/products/esl/system_debug/vista/index.cfm (checked 02/12/08). Mentor Graphics Corp., Oregon, USA, 2007.
- [87] MENTOR GRAPHICS. *ModelSim SE User’s Manual*. Mentor Graphics Corp., Wilsonville, USA, July 2007.
- [88] MEREDITH, M. OSCI General Update & Working Group Plans. In *Proc. 14th European SystemC User Group (ESCUG) Meeting* (Darmstadt, September 2006).
- [89] MISCHKALLA, F. Realisierung des EmViD Video-Prozessors auf dem XUP Virtex-II Pro FPGA-System. Diploma thesis, TU Braunschweig, Abt. E.I.S., February 2007.
- [90] MONTOREANO, M. Transaction Level Modeling using OSCI TLM 2.0. *OSCI TLM Working Group* (May 2007).
- [91] MÜLLER, W., ROSENSTIEL, W., AND RUF, J., Eds. *SystemC: Methodologies and Applications*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2003.
- [92] NAUMANN, A. Was Darwin wrong? Has design evolution stopped at the RTL level or will software and custom processors (or system-level design) extend Moore’s law? In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Nice, April 2007), p. 2.
- [93] NXP. *Nexperia homepage*: <http://www.nxp.com/products/nexperia/about> (checked 02/12/08). NXP Semiconductors, Eindhoven, 2007.
- [94] OCP-IP. *Open Core Protocol Specification 2.2*. Open Core Protocol International Partnership (OCP-IP), Beaverton, USA, 2006.
- [95] OCP-IP. *A SystemC OCP Transaction Level Communication Channel*. Available at: <http://www.ocpip.org/socket/systemc> (checked 02/12/08). Open Core Protocol International Partnership (OCP-IP), Beaverton, USA, February 2007.
- [96] OPEN CORE PROTOCOL INTERNATIONAL PARTNERSHIP (OCP-IP). Prosilog and OCP-IP Announce a Family of OCP Compliant AMBA and CoreConnect Bridges. *Press Release*, see <http://www.ocpip.org/pressroom/releases/2003/prosilog-081803> (checked 02/12/08) (August 2003).
- [97] OSCI. *SystemC 2.1 Language Reference Manual*. Available at <http://www.systemc.org> (checked 02/12/08). Open SystemC Initiative, April 2005.
- [98] OSCI. Requirements specification for TLM 2.0. Tech. rep., Open SystemC Initiative (OSCI) TLM Working Group, available at <http://www.systemc.org> (checked 02/12/08), June 2007.
- [99] OSCI. *SystemC Verification Standard Specification*. Available at <http://www.systemc.org> (checked 02/12/08). Open SystemC Initiative (OSCI) SystemC Verification Working Group, May 2004.

-
- [100] OSGi ALLIANCE. *About the OSGi Service Platform (white paper)*. Available at <http://www.osgi.org/Specifications/HomePage> (checked 02/12/08). OSGi Alliance, June 2007.
 - [101] OUADJAOUT, S., AND HOUZET, D. Generation of Embedded Hardware/Software from SystemC. *EURASIP Journal on Embedded Systems, Article ID 18526* (Hindawi Publishing Corp., Cairo, 2006).
 - [102] PAEPKE, T. Erweiterung des SystemC-Analyseframeworks DUST um eine interaktive Analyse von GreenBus-Modellen. Diploma thesis, TU Braunschweig, Abt. E.I.S., April 2007.
 - [103] PASRICHA, S., DUTT, N., AND BEN-ROMDHANE, M. Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration. In *Proc. Design Automation Conference (DAC)* (San Diego, USA, June 2004), pp. 113–118.
 - [104] PATRICK, J. B. Device & Register Framework: Technology & API Overview. *Internal presentation material* (Intel Corp., Phoenix, USA, October 2007).
 - [105] PCI SPECIAL INTEREST GROUP. *PCI Express Base Specification Revision 2.0*, available at: <http://www.pcisig.com/specifications/pciexpress> (checked 02/12/08). PCI-SIG, Beaverton, USA, December 2006.
 - [106] POYNTON, C. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers, San Francisco, USA, 2003.
 - [107] RIVERA, R., DAWES, B., AND ABRAHAMS, D. *Boost library homepage*: <http://www.boost.org> (checked: 02/12/08). Boost Developers Group, 2007.
 - [108] ROBERT BOSCH GMBH. *CAN Specification, Version 2.0*. Robert Bosch GmbH, Gerlingen, 1991.
 - [109] ROSE, A., SWAN, S., PIERCE, J., AND M. FERNANDEZ, J. Transaction Level Modeling in SystemC. Tech. rep., Open SystemC Initiative (OSCI) TLM Working Group, <http://www.systemc.org> (checked 02/12/08), 2005.
 - [110] ROSIN, M. Ein Framework zur Modellierung eingebetteter Videoverarbeitungs-Systeme mit SystemC. Study thesis, TU Braunschweig, Abt. E.I.S., August 2006.
 - [111] SANGIOVANNI-VINCENTELLI, A., AND MARTIN, G. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers* 18, 6 (November-December 2001), 23–33.
 - [112] SCHIRNER, G., AND DÖMER, R. Quantitative Analysis of Transaction Level Models for the AMBA Bus. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2006).
 - [113] SCHIRNER, G., AND DÖMER, R. Fast and Accurate Transaction Level Models using Result Oriented Modeling. In *Proc. Int. Conf. on Computer-Aided Design (ICCAD)* (San Jose, USA, November 2006).
 - [114] SCHRÖDER, C., AND KLINGAUF, W. GreenBus PCI Express User’s Guide. Tech. rep., TU Braunschweig, Abt. E.I.S., and GreenSocs Ltd., November 2007.
 - [115] SEMMELHACK, P. Exploring the Long Tail of Devices - How an open web services and hardware platform can enable the creation of truly personal consumer electronics. In *Proc. World Conference on Mass Customization & Personalization (MCPC)* (Cambridge, USA, October 2007).

- [116] SIEGMUND, R., AND MÜLLER, D. SystemC-SV: An Extension of SystemC for Mixed Multi-Level Communication Modeling and Interface-Based System Design. In *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, March 2001), pp. 26–33.
- [117] SPIRIT CONSORTIUM. SPIRIT User Guide v1.1. *SPIRIT Schmea Working Group* (June 2005).
- [118] SPÖR, C. Generische Software-Generierung aus SystemC. Diploma thesis, TU Braunschweig, Abt. E.I.S., March (expected date) 2008.
- [119] ST MICROELECTRONICS. *TAC: Transaction Accurate Communication*. Available at: <http://www.greensocs.com/TACPackage> (checked 02/12/08). GreenSocs Ltd., Cambridge, UK, 2005.
- [120] ST MICROELECTRONICS. *Nomadik - Open Multimedia Platform for Next-generation Mobile Devices*. Available at www.st.com/stonline/products/families/mobile/processors/processor-sprod.htm (checked 02/12/08). ST Microelectronics, Geneva, 2007.
- [121] STEINHÄUSER, I. Ein IP-Core zur Video-Digitalisierung mit Virtex-II-Pro-FPGAs. Diploma thesis, TU Braunschweig, Abt. E.I.S., July 2007.
- [122] STEINHÄUSER, I. Ethernet-Anbindung von Xilinx-Plattform-FPGAs. Study thesis, TU Braunschweig, Abt. E.I.S., April 2007.
- [123] STORK, H. Structuring Process and Design for Future Mobile Communication Devices. In *Keynote Presentation at Design Automation Conference (DAC)* (San Francisco, USA, 2006).
- [124] STRANO, G., TIRALONGO, S., AND PISTRITTO, C. OCP/STBus Plug-In Methodology. In *Proc. Global Signal Processing Conference (GSPx)* (Santa Clara, USA, September 2004).
- [125] STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley Longman, Old Tappan, USA, February 2000.
- [126] TEXAS INSTRUMENTS. *OMAP 3 family of multimedia applications processors*. Available at <http://focus.ti.com/pdfs/wtbu/ti-omap3family.pdf> (checked 02/12/08). Texas Instruments Inc., Dallas, USA, 2007.
- [127] USB IMPLEMENTORS FORUM. *Universal Serial Bus Specification*, available at <http://www.usb.org/developers/docs> (checked 02/12/08). Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, April 2000.
- [128] W3C. *XML Schema homepage*: <http://www.w3.org/XML/Schema> (checked 02/12/08). World Wide Web Consortium (W3C), 2004.
- [129] WAGNER, F. R., CESARIO, W. O., CARRO, L., AND JERRAYA, A. A. Strategies for the integration of hardware and software IP components in embedded systems-on-chip. *Integration, the VLSI Journal; special issue on IP and design reuse* 37, 4 (September 2004), 223–252.
- [130] WEISER, M. The Computer for the 21st Century. *Scientific American* 256, 3 (September 1991), 94–104.
- [131] XILINX. *Virtex-II Pro Platform FPGAs: Advance Product Specification*. Xilinx Inc., San Jose, USA, 2003.
- [132] XILINX. *BFM Simulation in Platform Studio*. Xilinx Inc., San Jose, USA, November 2005.
- [133] XILINX. *Processor IP Reference Guide*. Xilinx Inc., San Jose, USA, June 2005.

- [134] XILINX. *Xilinx University Program Virtex-II Pro Development System*. Xilinx Inc., San Jose, USA, March 2005.
- [135] XILINX. *XST User Guide*. Xilinx Inc., San Jose, USA, October 2006.
- [136] XILINX. *EDK Concepts, Tools, and Techniques*. Xilinx Inc., San Jose, USA, September 2007.
- [137] YAKOVLEV, A., MARCHAL, P., AND GINOSAR, R. Future Interconnects and Networks on Chip. In *Special Workshop during the Design, Automation and Test in Europe Conference (DATE)*. Available at <http://async.org.uk/noc2006> (checked 02/12/08) (Munich, March 2006).
- [138] YU, H., DÖMER, R., AND GAJSKI, D. Embedded Software Generation from System Level Design Languages. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)* (Yokohoma, Japan, January 2004), pp. 463–468.
- [139] YU, H., GERSTLAUER, A., AND GAJSKI, D. RTOS Scheduling in Transaction Level Models. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Newport Beach, USA, October 2003), pp. 31–36.
- [140] ZECHNER, A. Hardware-Synthese mit SytemC: Analyse von Methoden und Werkzeugen am Beispiel eines JPEG-Encoders. Study thesis, TU Braunschweig, Abt. E.I.S., May 2005.
- [141] ZWERENZ, K. Visualisierung und grafische Analyse von GreenBus-basierten System-on-Chip-Modellen. Diploma thesis, TU Braunschweig, Abt. E.I.S., April 2007.

Abbreviations

AHB Advanced High-performance Bus	HDL Hardware Description Language
AMBA Advanced Microcontroller Bus Architecture	HPC Hardware Procedure Call
API Application Programming Interface	HW Hardware
ASIC Application-Specific Integrated Circuit	IDE Integrated Development Environment
AST Abstract Syntax Tree	IP Intellectual Property
AXI Advanced eXtensible Interface	IMC Interface Method Call
BA Bus-Accurate (abstraction level)	ISA Industry Standard Architecture
BRAM Block RAM	LRM Language Reference Manual
CA Communication Architecture	MoC Model of Computation
CAFM Communication Architecture Functional Model	MPSoC Multi-Processor SoC
CAN Controller Area Network	MRMD Multiple Request Multiple Data
CC Cycle-Count Accurate (abstraction level)	NoC Network-on-Chip
CCATB Cycle-count Accurate At The Boundaries - UC Irvine	OCCN On-Chip Communication Network
DCR Device Control Register Bus	OCP Open Core Protocol
DMA Direct Memory Access	OMAP Open Multimedia Applications Protocol
DSP Digital Signal Processor	OPB On-chip Peripheral Bus
DUST Dynamic and Universal SystemC Transaction Debugger	OSCI Open SystemC Initiative
EDA Electronic Design Automation	OSSS Oldenburg System Synthesis Subset
EDK Embedded Development Kit	PCI Peripheral Components Interconnect
EmViD Embedded Video Digestion	PCIe PCI Express
ESL Electronic System Level	PDU Protocol Data Unit
FIFO First-In First-Out	PE Processing Element
FPGA Field Programmable Gate Array	PEQ Payload Event Queue
GC GreenConfig	POD Plain Old Datatype
GP Generic Protocol	PV Programmers View (abstraction level)
HAL Hardware Abstraction Layer	PV+T Programmers View with time
	PLB Processor Local Bus
	ROM Result Oriented Modeling

RTL Register Transfer Logic

RTOS Real-time operating system

RTTI Run-time type information

SHIP SystemC High-level Interface Protocol

SLDL System-level design language

SCML SystemC Model Library

SCV SystemC Verification Standard

SoC System-on-Chip

SQL Structured Query Language

SRMD Single Request Multiple Data

STL Standard Template Library

SW Software

TA Technology Adapter

TAC Transaction Accurate Communication

TAQ Transaction, Atom, Quark

TC Transaction Container

TLM Transaction-Level Modeling

TRAIN Transaction Interchange

VCD Value Change Dump

VIC Video IP Core (EmViD project)

VTL Virtual Transaction Layer

XML eXtensible Markup Language

Index

- Abstraction, 7, 13, 22, 37, 45
 - Formalism, 41
- Adapter, 16
- Address map, 81
- AMBA, 27, 38, 119
- API, 21
- Application Programming Interface, *see* API
- Arbitration, 84, 120
- Architecture mapping, 14
- Atom, 37, 58

- BA, 37, 73, 83
- Back annotation, 141
- Blocking, 30, 136
- Bridge, 122
- Bus accurate, *see* BA

- CAFM, 15, 30, 36
- CC, 73, 83
- CCATB, 23, 27
- Channel, 11, 14
- Communication delay, 46, 50
- Configurable parameter, 103
- Copy at slave, 48
- CoreConnect, 21, 27, 38, 45, 89, 119

- DCR, *see* CoreConnect
- DUST, 106

- Electronic System Level, *see* ESL
- ESL, 19
- Event, 12
 - Simulator, 12

- FPGA, 142

- Generic Protocol, 56
- GreenBus, 30, 33
 - Accuracy, 90, 98
 - Compatibility, 68
 - Extension, 62
 - Layers, 35
 - Performance, 92
 - Port, 48
 - Protocol, 56
 - Router, 36, 81
 - Transaction Container, 56
 - Transport, 43, 58
- GreenControl, 101

- LEGO, 3, 115

- Master, 54
- Memory pooling, 72
- Message passing, 136
- Module, 11
- MRMD, 24

- Network-on-Chip, *see* NoC
- NoC, 95, 122
- Nonblocking, 31
- Notification, 50

- OCCN, 27
- OCP, 20, 26, 41, 77, 95, 122, 140
- OMAP-3, 19
- OPB, *see* CoreConnect
- Open Core Protocol, *see* OCP
- Open SystemC Initiative, *see* OSCI
- OSCI, 10
 - TLM, 25, 26
- OSSS, 27

- Payload Event Queue, *see* PEQ
- PCI Express, 95, 121
- PE, 12
- PEQ, 49
- Performance analysis, 101
- Platform-based design, 20
- PLB, *see* CoreConnect
- Pointer, 57
- Port, 11
 - Generic, 53
 - Hierarchical, 52
- Processing Element, *see* PE
- Programmers View, *see* PV
- Protocol, 14
 - Simulation, 87
- PV, 22, 73, 83

- Quark, 38
- ROM, 28
- Router, 81
- Scheduling, 84
- SHIP, 136, 140
- Slave, 54
- SoC, 19
- Software, 144
- SRMD, 24
- STBus, 120
- Synchronous, 136
- System-level design, 19
- System-on-Chip, *see* SoC
- SystemC, 7, 10
 - sc_event, 12
 - SC_METHOD, 11
 - SC_MODULE, 11
 - SC_THREAD, 11
 - Simulator, 12
- SystemC^{SV}, 26
- TAC, 21, 26, 77
- TAQ, 37
- Testbench, 106
- TLM, 7
 - Fabric, 20
 - Framework, 26
 - Model, 12
 - Timed, 46, 83
 - Untimed, 44, 83
- TRAIN, 142
- Transaction Accurate Communication, *see* TAC
- Transaction Container, 43, 56
- Transaction Level Modeling, *see* TLM
- Transactions, atoms, quarks, *see* TAQ
- User API, 35, 52, 69, 105, 138
 - Design patterns, 123
 - State machines, 127

List of Figures

1.1	Play?	3
1.2	Expected advantages of TLM over RTL design	5
2.1	Embedded system design with TLM	8
2.2	TLM abstraction levels in terms of timing accuracy (from [18])	9
2.3	Structure of SystemC	10
2.4	SystemC TLM with two PEs connected by a channel	11
2.5	SystemC event-driven simulator	13
2.6	Abstract channels may, but need not necessarily reflect the behavior of a physical communication architecture	14
2.7	Example TLM with heterogeneous PEs connected to a CAFM	17
3.1	Texas Instruments OMAP-3 system-on-chip (from [126])	19
3.2	Bus Transactions	23
4.1	Layered communication approach	35
4.2	Example system model with both point-to-point channels and a CAFM based on GREENBUS	36
4.3	Overlapping atoms of concurrent PLB transactions	38
4.4	Simulation of RTL transactions with atoms and quarks	43
4.5	An atom's life cycle	43
4.6	GREENBUS transaction ³	44
4.7	Representation of a PLB transaction with atoms and quarks	45
4.8	Delay model using the GREENBUS low-level transport interface	47
4.9	Nonblocking basic port communication	49
4.10	Blocking TLM port communication	49
4.11	Processing of payload events	50
4.12	Effects of immediate (non-delayed) data transfers	52
4.13	Port inheritance class diagram	53
4.14	Master-slave communication via hierarchical GREENBUS ports	55
4.15	Possible implementation of the generic transaction container ⁴	57
4.16	Write transaction using the generic protocol	60
4.17	Read transaction using the generic protocol	61
4.18	Interoperability example: mixing three transaction container types in one model	62
4.19	Class <code>PCIELightTransaction</code> inherits from <code>GenericTransaction</code> and extends it by two new quarks	63
4.20	Transaction container extension using virtual inheritance	65
4.21	Simulation performance comparison of different demo platforms, using virtual vs. non-virtual transaction container class hierarchy	66
4.22	Transaction container extension using static cast hierarchy	67
4.23	Using the generic protocol as compatibility interface for encrypted IP cores	69
4.24	Data and control flow using the generic API and the generic protocol	70
4.25	Simulation performance for point-to-point transactions via GREENBUS at BA abstraction using different transaction container creation techniques	72

4.26	Comparison of simulation performance for 1 mio. transactions of a varying number of masters to a slave via GREENBUS at BA abstraction using different PEQ implementations	73
4.27	PV - BA - CC performance comparison of GREENBUS point-to-point writes	74
4.28	Impact of request acknowledge delay on GREENBUS performance	74
4.29	Example of BA \Leftrightarrow CC mixed-mode communication: TAC to OCP-tl1 writes	74
4.30	BA communication between a CCATB accurate and a CC accurate user API	75
4.31	PV performance: comparison of TAC transactions over GREENBUS with TAC transactions over ST's TAC fabric	76
4.32	OCP-tl1 simulation performance of GREENBUS compared to the original OCP-IP channel	76
5.1	Bus functional simulation subsystem	82
5.2	Transaction between a PV master and a PV slave over the generic router	83
5.3	Transaction between a CCATB accurate master and a CCATB accurate slave over the generic router	84
5.4	Mixed-mode transaction between a PV master and a CC accurate slave over the generic router	85
5.5	Combinatorial arbitration in RTL and TLM models	86
5.6	OPB protocol class state machine	88
5.7	Evaluation of GREENBUS / PLB simulation quality	91
5.8	Waveform output of the GREENBUS PLB bus functional model at CC abstraction	91
5.9	GREENBUS PLB simulation performance relative to Xilinx RTL bus functional model performance	92
5.10	GREENBUS PLB simulation performance compared to IBM SystemC and Xilinx VHDL bus functional models	93
5.11	GREENBUS PLB simulation performance in dependence of bus load	94
5.12	GREENBUS PLB simulation performance in dependence of PE number	94
5.13	PCI Express example architecture	96
5.14	PV transaction performance (write bursts over a PCIe switch)	97
5.15	Write burst simulation performance of an OCP NoC model depending on number of hops (length of transaction path in terms of NoC switches)	98
5.16	Communication delays applied by a memory controller PE at different abstractions	99
5.17	Simulation of a stalled transaction at different abstractions	100
6.1	IP block, ESL tool, and GREENCONTROL library	102
6.2	GREENCONTROL framework with service plugins	103
6.3	DUST analysis framework for SystemC	107
6.4	Design structure visualization and parameter monitoring	108
6.5	Message sequence chart view	108
6.6	Communication between frontend and backend	109
6.7	ModelSim displaying GREENBUS transactions using SCV	110
A.1	CoreConnect architecture	120
B.1	Usage of <code>MSBytesValid</code> in generic protocol transactions	127
B.2	State diagram for master user API; write transaction with fixed burst length (both single-beat and SRMD)	129
B.3	State diagram for master user API; read transaction with fixed burst length (both single-beat and SRMD)	130
B.4	State diagram for slave user API; write transaction with fixed burst length (both single-beat and SRMD)	131

B.5	State diagram for slave user API; read transaction with fixed burst length (both single-beat and SRMD)	132
C.1	Systematic embedded system design flow	135
C.2	SHIP communication	137
C.3	EmViD abstract PEs use SHIP communication	137
C.4	SHIP objects used in EmViD for video frame processing and transport	138
C.5	Gesture recognition pipeline based on SHIP PEs	139
C.6	Graphical composition of EmViD systems	139
C.7	The dilation component has been refined to an implicit SM with OCP-tl1 ports and attached BRAM block. The rest of the system is still at an untimed level of abstraction.	141
C.8	The dilation component has been refined to an explicit SM with pin accurate communication.	141
C.9	Example of measuring the computation time of the refined dilation component. The waveform output was generated with GREENCONTROL. You can see two OCP transactions, one for receiving a video frame on the slave port, and one for sending the frame to the master port, after applying the dilatation algorithm. The marked section of 199220 ns is the measured delay at 100 MHz clock speed.	142
C.10	TRAIN architecture overview	143
C.11	EmViD cores connected to a Processor Local Bus (PLB) using accessors	143
C.12	EmViD architecture with a shared BRAM, using both OPB and PLB accessors	143
C.13	Two colored cups detected by the video processor model. The left image shows the input video with superimposed detection information. The middle image shows the intermediate result after color matching, erosion, and dilation have been applied. The right image shows the detected regions after the labeling and region detection phase.	144
C.14	Motion traces recorded by the gesture recognition PE for the detected objects (here, the two hands and the head).	145
C.15	Golden model of the gesture recognition system	145
C.16	The IP cores video digitizer, display, and DDR memory have been integrated into the TLM model bottom-up.	147
C.17	EmViD pipeline architecture	147
C.18	Data flow in the pipeline architecture	148
C.19	EmViD bus architecture	149
C.20	Data flow in the bus architecture using fixed ascending priorities	149
C.21	The graph shows the predicted number of clock cycles per frame for region detection, using the differently abstract TLM models	152
C.22	EmViD SoC floorplan for the Virtex-II Pro FPGA V2P30	153
C.23	Our EmViD testbench in the lab	154

List of Tables

3.1	Characteristics of reviewed TLM frameworks	29
4.1	Detailed user-view requirements for GREENBUS	34
4.2	Communication layers in GREENBUS-based TLM models	36
4.3	Standard quark set	39
4.4	Mapping of PLB and AXI onto GREENBUS quarks and atoms	40
4.5	Abstraction / time resolution relationship	42
4.6	GREENBUS ports	53
4.7	Generic protocol API	58
4.8	GREENBUS CC transactions performance with OCP-tl1 user APIs: results of the first implementation (presented in [74]) compared to the final version	71
4.9	Implementation of the user-view requirements from table 4.1 in GREENBUS	78
5.1	GREENBUS PLB simulation performance of 512 byte writes relative to RTL, in comparison to the performance of AMBA models from Schirner et al. [112]	92
5.2	Fulfillment of user-view requirements from table 4.1 by the generic router	100
6.1	This spreadsheet shows the capabilities of the reviewed configuration libraries for SystemC and compares them with the GREENCONTROL framework	104
6.2	Example of a GREENCONTROL transaction container	105
6.3	Fulfillment of user-view requirements from table 4.1 by GREENCONTROL	110
C.1	SHIP-based video processing components for gesture recognition	139
C.2	HW/SW partitioning of EmViD gesture recognition system-on-chip	146
C.3	EmViD execution traces	150
C.4	Simulation performance of EmViD models at different abstraction levels	151
C.5	Virtex-II Pro FPGA resource allocation of the EmViD components	152

List of Publications

Book Chapters

Günzel, R., Klingauf, W., Aldis, J., Combinatorial Dependencies in Transaction Level Models, *to appear in* ‘Embedded Systems Specification and Design Languages’, Springer Verlag, March 2008.

Grosse, D., Peraza, H., Klingauf, W., Drechsler, R., Measuring the Quality of a SystemC Testbench by using Code Coverage Techniques, *to appear in* ‘Embedded Systems Specification and Design Languages’, Springer Verlag, March 2008.

Conference Papers

Klingauf, W., Günzel, R., and Schröder, C., Embedded Software Development on Top of Transaction Level Models, *Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Salzburg, September 2007.

Grosse, D., Peraza, H., Klingauf, W., Drechsler, R., Measuring the Quality of a SystemC Testbench by using Code Coverage Techniques, *Forum on Specification and Design Languages (FDL)*, Barcelona, September 2007.

Burton, M., Aldis, J., Günzel, R., Klingauf, W., Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified, *Forum on Specification and Design Languages (FDL)*, Barcelona, September 2007.

Günzel, R., Klingauf, W., Aldis, J., Combinatorial Dependencies in Transaction Level Models, *Forum on Specification and Design Languages (FDL)*, Barcelona, September 2007.

Klingauf, W., Geffken, M., Design Structure Analysis and Transaction Recording in SystemC Designs: A Minimal-Intrusive Approach, *Forum on Specification and Design Languages (FDL)*, Darmstadt, September 2006.

Klingauf, W., Günzel, R., Bringmann, O., Parfuntseu, P., Buron, M., GreenBus – A Generic Interconnect Fabric for Transaction Level Modelling, *43rd Design Automation Conference (DAC)*, San Francisco, July 2006.

Klingauf, W., Gädke, H., Günzel, R., TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC, *Design, Automation and Test in Europe Conference (DATE)*, München, March 2006.

Klingauf, W., Günzel, R., From TLM to FPGA: Rapid Prototyping with SystemC and Transaction Level Modeling, *IEEE Int. Conf. on Field-Programmable Technology (FPT)*, Singapore, December 2005.

Klingauf, W., Systematic Transaction Level Modeling of Embedded Systems with SystemC, *Design, Automation and Test in Europe Conference (DATE)*, München, March 2005.

Klingauf, W., Witte, L., Golze, U., Performance Optimization of Embedded Java Applications by a C/Java-hybrid Architecture, *Global Signal Processing Conference (GSPx)*, Santa Clara, September 2004.

Klingauf, W., Telkamp, G., Böhme, H., Java-basierte Benutzeroberflächen für extrem kompakte eingebettete Systeme, *Entwicklerforum Embedded Betriebssysteme*, München, October 2003.

Böhme, H., Klingauf, W., Telkamp, G., JControl – Rapid Prototyping und Design Reuse mit Java, *11. E.I.S. Workshop*, Erlangen, March 2003.

Best Paper Awards

FDL'07 Best Paper Award – Grosse, D., Peraza, H., Klingauf, W., Drechsler, R., 'Measuring the Quality of a SystemC Testbench by Using Code Coverage Techniques', Forum on Specification and Design Languages (FDL), Barcelona, September 2007.

DATE'05 Best Paper Interactive Presentation Award – Klingauf, W., 'Systematic Transaction Level Modeling of Embedded Systems with SystemC', Design, Automation and Test in Europe Conference (DATE), München, March 2005.

Curriculum Vitae

Name: Wolfgang Heinrich Klingauf
Born: 15 April 1976 in Bonn, Germany
Marital Status: Married

Education

1982 - 1986 Grundschole in Darmstadt
1986 - 1989 Ludwig-Georgs-Gymnasium in Darmstadt
1989 - 1995 Wilhelm Gymnasium in Braunschweig
15 May 1995 Abitur at Wilhelm Gymnasium
1995 - 1996 Zivildienst at Deutsches Rotes Kreuz in Wolfenbüttel,
Apprenticeship as Rettungssanitäter (paramedic)
1996 - 2002 Study of Computer Science at Technical University Braunschweig
21 August 2002 Diploma in Computer Science
Graduated 'Mit Auszeichnung' (with Honors)
Thesis title: Integration des Hardware/Software-Systems Colibree in ein
Local Operating Network (LON)
Advisors: Prof. Ulrich Golze and Prof. Rolf Ernst

Work Experience: Academic

2002 - 2008 Research Assistant (WiMi)
Dept. Integrated Circuit Design (Prof. U. Golze)
Technical University of Braunschweig
1999 - 2002 Student Assistant (HiWi)
Institute of Operating Systems and Computer Networks
(Prof. M. Zitterbart), Technical University of Braunschweig

Work Experience: Commercial

2002 Design Engineer
DOMOLOGIC Home Automation GmbH, Braunschweig

